Linux 2.4 stateful firewall design

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. About this tutorial

Should I take this tutorial?

This tutorial shows you how to use netfilter to set up a powerful Linux stateful firewall. All you need is an existing Linux system that's currently using a Linux 2.4 kernel. A laptop, workstation, router or server with a Linux 2.4 kernel will do.

You should be reasonably familiar with standard network terminology like IP addresses, source and destination port numbers, TCP, UDP and ICMP, etc. By the end of the tutorial, you'll understand how Linux stateful firewalls are put together and you'll have several example configurations to use in your own projects.

About the author

For technical questions about the content of this tutorial, contact the author, Daniel Robbins, at *drobbins@gentoo.org* .

Residing in Albuquerque, New Mexico, Daniel Robbins is the President/CEO of *Gentoo Technologies, Inc.* , the creator of **Gentoo Linux**, an advanced Linux for the PC, and the **Portage** system, a next-generation ports system for Linux. He has also served as a contributing author for the Macmillan books *Caldera OpenLinux Unleashed*, *SuSE Linux Unleashed*, and *Samba Unleashed*. Daniel has been involved with computers in some fashion since the second grade, when he was first exposed to the Logo programming language as well as a potentially dangerous dose of Pac Man. This probably explains why he has since served as a Lead Graphic Artist at **SONY Electronic Publishing/Psygnosis**. Daniel enjoys spending time with his wife, Mary, and his new baby daughter, Hadassah.

Section 2. First steps

Defining our goal

In this tutorial, we're going to put together a Linux stateful firewall. Our firewall is going to run on a Linux laptop, workstation, server, or router; its primary goal is to allow only certain types of network traffic to pass through. To increase security, we're going to configure the firewall to drop or reject traffic that we're not interested in, as well as traffic that could pose a security threat.

Getting the tools

Before we start designing a firewall, we need to do two things. First, we need to make sure that the "iptables" command is available. As root, type "iptables" and see if it exists. If it doesn't, then we'll need to get it installed first. Here's how: head over to *http://netfilter.samba.org* and grab the most recent version of iptables.tar.gz (currently iptables-1.1.2.tar.gz) you can find. Then, install it by typing in the following commands (output omitted for brevity):

```
# tar xzvf iptables-1.1.2.tar.gz
# cd iptables-1.1.2
# make
# make install
```

Kernel configuration, Part 1

Once installed, you should have an "iptables" command available for use, as well as the handy iptables man page ("man iptables"). Great; now all we need is to make sure that we have the necessary functionality built into the kernel. This tutorial assumes that you compile your own kernels. Head over to /usr/src/linux, and type "make menuconfig" or "make xconfig"; we're going to enable some kernel network functionality.

Kernel configuration, Part 2

Under the "Networking options" section, make sure that you enable at least the following options:

```
<*> Packet socket
[*] Network packet filtering (replaces ipchains)
<*> Unix domain sockets
[*] TCP/IP networking
[*]   IP: advanced router
[*]   IP: policy routing
[*]     IP: use netfilter MARK value as routing key
```

```
[*]     IP: fast network address translation
[*]   IP: use TOS value as routing key
```

Then, under the "IP: Netfilter Configuration --- >" menu, enable *every option* so that we'll have full netfilter functionality. We won't use all the netfilter features, but it's good to enable them so that you can do some experimentation later on.

Kernel configuration, Part 3

There's one networking option under the "Networking options" category that you *shouldn't* enable: explicit congestion notification. Leave this option disabled:

```
[ ]   IP: TCP Explicit Congestion Notification support
```

If this option is enabled, your Linux machine won't be able to carry on network communications with 8% of the Internet. When ECN is enabled, some packets that your Linux box sends out will have the ECN bit set; however, this bit freaks out a number of Internet routers, so it's very important that ECN is disabled.

OK, now that the kernel's configured correctly for our needs, compile a new one, install it, and reboot. Time to start playing with netfilter :)

Firewall design basics

In putting together our firewall, the "iptables" command is our friend. It's what we use to interact with the network packet filtering rules in the kernel. We'll use the "iptables" command to create new rules, list existing rules, flush rules, and set default packet handling policies. This means that to create our firewall, we're going to enter a series of iptables commands, and here's the first one we're going to take a look at (please don't type this in just yet!)...

Firewall design basics, continued

```
# iptables -P INPUT DROP
```

You're looking at an almost "perfect" firewall. If you type in this command, you'll be incredibly well protected against any form of incoming malicious attack. That's because this command tells the kernel to drop all incoming network packets. While this firewall is extremely secure, it's a bit silly. But before moving on, let's take a look at exactly how this command does what it does.

Setting chain policy

An "iptables -P" command is used to set the default *policy* for a chain of packet filtering rules. In this example, iptables -P is used to set the default policy for the INPUT chain, a built-in chain of rules that's applied to every incoming packet. By setting the default policy to DROP, we tell the kernel that any packets that reach the end of the INPUT rule chain should be dropped (that is, discarded). And, since we haven't added any rules to the INPUT chain, all packets reach the end of the chain, and all packets are dropped.

Setting chain policy, continued

Again, by itself this command is totally useless. However, it demonstrates a good strategy for firewall design. We'll start by dropping all packets by default, and then gradually start opening up our firewall so that it meets our needs. This will ensure that our firewall is as secure as possible.

Section 3. Defining rules

A (small) improvement

In this example, let's assume that we're designing a firewall for a machine with two network interfaces, eth0 and eth1. The eth0 network card is connected to our LAN, while the eth1 network card is attached to our DSL router, our connection to the Internet. For such a situation, we could improve our "ultimate firewall" by adding one more line:

```
iptables -P INPUT DROP
iptables -A INPUT -i ! eth1 -j ACCEPT
```

This additional "iptables -A" line adds a new packet filtering rule to the end of our INPUT chain. After this rule is added, our INPUT chain consists of a single rule and a drop-by-default policy. Now, let's take a look at what our semi-complete firewall does.

Following the INPUT chain

When a packet comes in on any interface (lo, eth0, or eth1), the netfilter code directs it to the INPUT chain and checks to see if the packet matches the first rule. If it does, the packet is accepted, and no further processing is performed. If not, the INPUT chain's default policy is enforced, and the packet is discarded (dropped).

That's the conceptual overview. Specifically, our first rule matches all packets coming in from eth0 and lo, immediately allowing them in. Any packets coming in from eth1 are dropped. So, if we enable this firewall on our machine, it'll be able to interact with our LAN but be effectively disconnected from the Internet. Let's look at a couple of ways to enable Internet traffic.

Traditional firewalls, Part 1

Obviously, for our firewall to be useful, we need to selectively allow some incoming packets to reach our machine via the Internet. There are two approaches to opening up our firewall to the point where it is useful -- one uses static rules, and the other uses dynamic, stateful rules.

Traditional firewalls, Part 2

Let's take downloading Web pages as an example. If we want our machine to be able to download Web pages from the Internet, we can add a static rule that will always be true for every incoming http packet, regardless of origin:

```
iptables -A INPUT --sport 80 -j ACCEPT
```

Since all standard Web traffic originates from a source port of 80, this rule effectively allows our machine to download Web pages. However, this traditional approach, while marginally acceptable, suffers from a bunch of problems.

Traditional firewall bummers, Part 1

Here's a problem: while most Web traffic originates from port 80, some doesn't. So, while this rule would work most of the time, there would be rare instances where this rule wouldn't work. For example, maybe you've seen a URL that looks like this: "http://www.foo.com:81". This example URL points to a Web site on port 81 rather than the default port 80, and would be unviewable from behind our current firewall. Taking into account all these special cases can quickly turn a fairly secure firewall into swiss cheese and quickly fill our INPUT chain with a bunch of rules to handle the occasional oddball Web site.

Traditional firewall bummers, Part 2

However, the major problem with this rule is security related. Sure, it's true that only traffic with a source port of 80 will be allowed through our firewall. But the source port of a packet is not something that we have any control over, and it can be easily altered by an intruder. For example, if an intruder knew how our firewall were designed, he could bypass our firewall by simply making sure that all his incoming connections originated from port 80 on one of his machines! Because this static firewall rule is so easy to exploit, a more secure dynamic approach is needed. Thankfully, iptables and kernel 2.4 provide everything we need to enable dynamic, stateful filtering.

Section 4. Stateful firewalls

State basics

Rather than opening up holes in our firewall based on static protocol characteristics, we can use Linux's new connection tracking functionality to make firewall decisions based on the dynamic *connection state* of packets. Conntrack works by associating every packet with an individual bidirectional communications channel, or connection.

For example, consider what happens when you use telnet or ssh to connect to a remote machine. If you view your network traffic at the packet level, all you see is a bunch of packets zipping from one machine to another. However, at a higher level, this exchange of packets is actually a bidirectional communications channel between your local machine and a remote machine. Traditional (old-fashioned) firewalls only look at the individual packets, not recognizing that they're actually part of a larger whole, a connection.

Inside conntrack

That's where connection tracking technology comes in. Linux's conntrack functionality can "see" the higher-level connections that are taking place, recognizing your ssh session as a single logical entity. Conntrack can even recognize UDP and ICMP packet exchanges as logical "connections", even though UDP and ICMP are connectionless in nature; this is very helpful because it allows us to use conntrack to handle ICMP and UDP packet exchanges.

If you've already rebooted and are using your new netfilter-enabled kernel, you can view a list of active network connections that your machine is participating in by typing "cat /proc/net/ip_conntrack". Even with no firewall configured, Linux's conntrack functionality is working behind the scenes, keeping track of the connections that your machine is participating in.

The NEW connection state, Part 1

Conntrack doesn't just recognize connections, it also classifies every packet that it sees into one of four connection states. The first state that we're going to talk about is called NEW. When you type "ssh remote.host.com", the initial packet or burst of packets that originate from your machine and are destined for remote.host.com are in the NEW state. However, as soon as you receive even just a single reply packet from remote.host.com, any further packets you send to remote.host.com as part of this connection aren't considered NEW packets anymore. So, a packet is only considered NEW when it's involved in establishing a new connection, and no traffic has yet been received from the remote host (as part of this particular connection, of course).

The NEW connection state, Part 2

I've described outgoing NEW packets, but it's also very possible (and common) to have incoming NEW packets. Incoming NEW packets generally originate from a remote machine, and are involved in initiating a connection *with you*. The initial packet(s) your Web server receives as part of a HTTP request would be considered incoming NEW packets; however, once you reply to just a single incoming NEW packet, any additional packets you receive that are related to this particular connection are no longer in the NEW state.

The ESTABLISHED state

Once a connection has seen traffic in both directions, additional packets relating to this connection are considered to be in an ESTABLISHED state. The distinction between NEW and ESTABLISHED is an important one, as we'll see in a minute.

The RELATED state

The third connection state category is called RELATED. RELATED packets are those that are starting a new connection, but are related to another currently existing connection. The RELATED state can be used to regulate connections that are part of a multi-connection protocol, such as ftp, as well as error packets related to existing connections (such as ICMP error packets related to an existing connection).

The INVALID state

Finally, there are INVALID packets -- those that can't be classified into one of the above three categories. It's important to note that if a packet is considered INVALID, it isn't automatically discarded; it's still up to you to insert the appropriate rules and set chain policy so that they're handled correctly.

Adding a stateful rule

OK, now that we have a good understanding of connection tracking, it's time to take a look at a single additional rule that transforms our non-functional firewall into something quite useful:

```
iptables -P INPUT DROP
iptables -A INPUT -i ! eth1 -j ACCEPT
iptables -A INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT
```

How the rule works

This single rule, when inserted at the end of our existing INPUT chain, will allow us to establish connections with remote machines. It works as follows. Let's say we want to ssh over to remote.host.com. After typing "ssh remote.host.com", our machine sends out a packet to initiate the connection. This particular packet is in the NEW state, and our firewall allows it out, because we're only blocking packets coming *in* to our firewall, not going out.

When we get a reply packet from remote.host.com, this packet trickles through our INPUT chain. It doesn't match our first rule (since it comes in on eth1), so it moves on to our next, and final rule. If it matches this rule, it will be accepted, and if it doesn't, it will fall off the end of the INPUT chain and the default policy will be applied to the packet (DROP). So, is this incoming reply packet accepted or dropped on the floor?

How the rule works, continued

Answer: accepted. When the kernel inspects this incoming packet, it first recognizes that it's part of an already existing connection. Then, the kernel needs to decide whether this is a NEW or ESTABLISHED packet. Since this is an incoming packet, it checks to see if this connection has had any outgoing traffic, and finds that it has (our initial NEW packet that we sent out). Therefore, this incoming packet is categorized as ESTABLISHED, as are any further packets we receive or send that are associated with this connection.

Incoming NEW packets

Now, let's consider what happens if someone on a remote machine tries to ssh in to *us*. The initial packet we receive is classified as NEW, and doesn't match rule 1, so it advances to rule 2. Because this packet isn't in an ESTABLISHED or RELATED state, it falls off the end of the INPUT chain and the default policy, DROP, is applied. Our incoming ssh connection request is dropped to the floor without so much as a reply (or TCP reset) from us.

A near-perfect firewall

So, what kind of firewall do we have so far? An excellent one for a laptop or a workstation where you don't want anyone from the Internet connecting *to you*, but where you need to connect to sites on the Internet. You'll be able to use Netscape, konqueror, ftp, ping, perform DNS lookups, and more. Any connection that you initiate will get back in through the firewall. However, any unsolicited connection that comes in from the Internet will be dropped, unless it's related to an existing connection that you initiated. As long as you don't need to provide any network services to the outside, this is a near-perfect firewall.

A basic firewall script

Here's a simple script that can be used to set up/tear down our first basic workstation firewall:

```
#!/bin/bash
# A basic stateful firewall for a workstation or laptop that isn't running any
# network services like a web server, SMTP server, ftp server, etc.

if [ "$1" = "start" ]
then
        echo "Starting firewall..."
        iptables -P INPUT DROP
        iptables -A INPUT -i ! eth1 -j ACCEPT
        iptables -A INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT
elif [ "$1" = "stop" ]
then
        echo "Stopping firewall..."
        iptables -F INPUT
        iptables -P INPUT ACCEPT
fi
```

Using the script

Using this script, you can bring down the firewall by typing "./firewall stop", and bring it back up again by typing "./firewall start". To bring down the firewall, we flush our rules out of the INPUT chain with a "iptables -F INPUT", and then switch the default INPUT policy back to ACCEPT with a "iptables -P INPUT ACCEPT" command. Now, let's look at a bunch of improvements that we can make to our existing workstation firewall. Once I've explained every improvement, I'll present a final workstation firewall script. Then, we'll start customizing our firewall for servers.

Section 5. Stateful improvements

Explicitly turn off ECN

I mentioned earlier that it's important to turn off ECN (explicit congestion notification) so that Internet communications will work properly. While you may have disabled ECN in the kernel per my suggestion, it's possible that in the future, you'll forget to do so. Or, possibly, you'll pass your firewall script along to someone who has ECN enabled. For these reasons, it's a good idea to use the /proc interface to explicitly disable ECN, as follows:

```
if [ -e /proc/sys/net/ipv4/tcp_ecn ]
then
        echo 0 > /proc/sys/net/ipv4/tcp_ecn
fi
```

Forwarding

If you're using your Linux machine as a router, then you'll want to enable IP forwarding, which will give the kernel permission to allow packets to travel between eth0 and eth1, and vice versa. In our example configuration, where eth0 is connected to our LAN, and eth1 is connected to the Internet, enabling IP forwarding is a necessary step in allowing our LAN to connect to the Internet via our Linux box. To enable IP forwarding, use this line:

```
echo 1 > /proc/sys/net/ipv4/ip_forward
```

Handling rejection, Part 1

So far, we've been dropping all unsolicited traffic coming in from the Internet. While this is an effective way to deter unwanted network activity, it does have some drawbacks. The biggest problem with this approach is that it's easy for an intruder to detect that we're running a firewall, since our machine isn't replying with the standard TCP reset and ICMP port-unreachable responses -- the responses that a normal machine would send back to indicate a failed connection attempt to a non-existent service.

Handling rejection, Part 2

Rather than let potential intruders know that we're running a firewall (and thus tip them off to the fact that we may be running some valuable services that they can't get to), it would be to our advantage to make it appear as if we aren't running any services at all. By adding these two rules to the end of our INPUT chain, we can successfully accomplish this task:

```
iptables -A INPUT -p tcp -i eth1 -j REJECT --reject-with tcp-reset
iptables -A INPUT -p udp -i eth1 -j REJECT --reject-with icmp-port-unreachable
```

Our first rule takes care of correctly zapping TCP connections, while the second handles UDP. With these two rules in place, it becomes very difficult for an intruder to detect that we're actually running a firewall; hopefully, this will cause the intruder to leave our machine and search for other targets with more potential for abuse.

Handling rejection, Part 3

In addition to making our firewall more "stealthy", these rules also eliminate the delay involved in connecting to certain ftp and irc servers. This delay is caused by the server performing an ident lookup to your machine (connecting to port 113) and eventually (after about 15 seconds) timing out. Now, our firewall will return a TCP reset and the ident lookup will fail immediately instead of retrying for 15 seconds (while you're patiently waiting for a response from the server).

Spoof protection

In many distributions, when the network interface(s) are brought up, several old ipchains rules are also added to the system. These special rules were added by the creators of the distribution to deal with a problem called spoofing, in which the source address of packets have been tweaked so that they contains an invalid value (something that script kiddies do). While we can create similar iptables rules that will also block spoofed packets, there's an easier way. These days, the kernel has the built-in ability to dropped spoofed packets; all we need to do is enable it via a simple /proc interface. Here's how.

Spoof protection, continued

```
for x in lo eth0 eth1
do
        echo 1 > /proc/sys/net/ipv4/conf/${x}/rp_filter
done
```

This shell script will tell the kernel to drop any spoofed packets on interfaces lo, eth0, and eth1. You can either add these lines to your firewall script, or add them to the script that brings up your lo, eth0, and eth1 interfaces.

Masquerading

NAT (network address translation) and IP masquerading, while not directly related to firewalls,

are often used in conjunction with them. We're going to look at two common NAT/masquerading configurations that you may need to use. This first rule would take care of situations where you have a dialup link to the Internet (ppp0) that uses a dynamic IP:

```
iptables -t nat -A POSTROUTING -o ppp0 -j MASQUERADE
```

If you're in this situation, you'll also want to convert my firewall scripts so that all references to "eth1" (our example DSL router) are changed to "ppp0". And it's perfectly fine to add firewalling rules that refer to "ppp0" when the ppp0 interface doesn't yet exist. As soon as ppp0 is up, everything will work perfectly. Make sure you enable IP forwarding as well.

SNAT

If you're using DSL to connect to the Internet, you probably have one of two possible configurations. One possibility is that your DSL router or modem has its own IP number and performs network address translation for you. If you're in this situation, you don't need Linux to perform NAT for you since your DSL router is taking care of it already.

However, if you want to have more control over your NAT functionality, you may want to talk to your ISP about configuring your DSL connection so that your DSL router is in "bridged mode". In bridged mode, your firewall becomes an official part of your ISP's network, and your DSL router transparently forwards IP traffic back and forth between your ISP and your Linux box without letting anyone know that it's there. It no longer has an IP number; instead, eth1 (in our example) sports the IP. If someone pings your IP from the Internet, they get a reply back from your Linux box, rather than your router.

SNAT, continued

With this kind of setup, you'll want to use SNAT (source NAT) rather than masquerading. Here's the line you should add to your firewall:

```
iptables -t nat -A POSTROUTING -o eth1 -j SNAT --to 1.2.3.4
```

In this example, eth1 should be changed to the ethernet interface connected directly to your DSL router, and 1.2.3.4 should be changed to your static IP (the IP of your ethernet interface). Again, remember to enable IP forwarding.

NAT issues

Fortunately for us, NAT and masquerading get along just fine with a firewall. When writing your firewall filtering rules, just ignore the fact that you're using NAT. Your rules should accept, drop, or reject packets based on their "real" source and destination addresses. The firewall

filtering code sees the original source address for a packet, and the final destination address. This is great for us, because it allows our firewall to continue working properly even if we temporarily disable NAT or masquerading.

Understanding tables

In the above NAT/masquerading examples, we're appending rules to a chain, but we're also doing something a bit different. Notice the "-t" option. The "-t" option allows us to specify the table that our chain belongs to. When omitted, the default table defaults to "filter". So, all our previous non-NAT related commands were modifying the INPUT chain that's part of the "filter" table. The "filter" table contains all the rules associated with accepting or rejecting packets, while the "nat" table (as you would assume) contains rules relating to network address translation. There are also other built-in iptables chains and they are described in detail in the iptables man page, as well as in Rusty's HOWTOs (see the Resources section at the end of this tutorial for links).

Our enhanced script

Now that we've taken a look at a bunch of possible enhancements, it's time to take a look at a second more flexible firewall up/down script:

```
#!/bin/bash

# An enhanced stateful firewall for a workstation, laptop or router that isn't
# running any network services like a web server, SMTP server, ftp server, etc.

#change this to the name of the interface that provides your "uplink"
#(connection to the Internet)

UPLINK="eth1"

#if you're a router (and thus should forward IP packets between interfaces),
#you want ROUTER="yes"; otherwise, ROUTER="no"

ROUTER="yes"

#change this next line to the static IP of your uplink interface for static SNAT,
#"dynamic" if you have a dynamic IP.  If you don't need any NAT, set NAT to "" to
#disable it.

NAT="1.2.3.4"

#change this next line so it lists all your network interfaces, including lo

INTERFACES="lo eth0 eth1"

if [ "$1" = "start" ]
then
```

```
        echo "Starting firewall..."
        iptables -P INPUT DROP
        iptables -A INPUT -i ! ${UPLINK} -j ACCEPT
        iptables -A INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT
        iptables -A INPUT -p tcp -i ${UPLINK} -j REJECT --reject-with tcp-reset
        iptables -A INPUT -p udp -i ${UPLINK} -j REJECT --reject-with icmp-port-u

        #explicitly disable ECN
        if [ -e /proc/sys/net/ipv4/tcp_ecn ]
        then
                echo 0 > /proc/sys/net/ipv4/tcp_ecn
        fi

        #disable spoofing on all interfaces
        for x in ${INTERFACES}
        do
                echo 1 > /proc/sys/net/ipv4/conf/${x}/rp_filter
        done

        if [ "$ROUTER" = "yes" ]
        then
                #we're a router of some kind, enable IP forwarding
                echo 1 > /proc/sys/net/ipv4/ip_forward
                if [ "$NAT" = "dynamic" ]
                then
                        #dynamic IP address, use masquerading
                        echo "Enabling masquerading (dynamic ip)..."
                        iptables -t nat -A POSTROUTING -o ${UPLINK} -j MASQUERADE
                elif [ "$NAT" != "" ]
                then
                        #static IP, use SNAT
                        echo "Enabling SNAT (static ip)..."
                        iptables -t nat -A POSTROUTING -o ${UPLINK} -j SNAT --to
                fi
        fi

elif [ "$1" = "stop" ]
then
        echo "Stopping firewall..."
        iptables -F INPUT
        iptables -P INPUT ACCEPT
        #turn off NAT/masquerading, if any
        iptables -t nat -F POSTROUTING
fi
```

Section 6. Stateful servers

Viewing rules

Before we start making customizations to our firewall so that it can be used on a server, I need to show you how to list your currently active firewall rules. To view the rules in the filter table's INPUT chain, type:

```
# iptables -v -L INPUT
```

The -v option gives us a verbose output, so that we can see the total packets and bytes transferred per rule. We can also look at our nat POSTROUTING table with the following command:

```
# iptables -t nat -v -L POSTROUTING
Chain POSTROUTING (policy ACCEPT 399 packets, 48418 bytes)
 pkts bytes target     prot opt in     out     source               destination
 2728  170K SNAT       all  --  any    eth1    anywhere             anywhere
```

Getting ready for service

Right now, our firewall doesn't allow the general public to connect to services on our machine because it only accepts incoming ESTABLISHED or RELATED packets. Because it drops any incoming NEW packets, any connection attempt is rejected unconditionally. However, by selectively allowing some incoming traffic to cross our firewall, we can allow the general public to connect to the services that we specify.

Stateful HTTP

While we want to accept *some* incoming connections, we probably don't want to accept every kind of incoming connection. It makes sense to start with a "deny by default" policy (as we have now) and begin opening up access to those services that we'd like people to be able to connect to. For example, if we're running a Web server, we'll allow NEW packets into our machine, as long as they are headed for port 80 (HTTP). That's all we need to do. Once we allow the NEW packets in, we've allowed a connection to be established. Once the connection is established, our existing rule allowing incoming ESTABLISHED and RELATED packets kicks in, allowing the HTTP connection to proceed unhindered.

Stateful HTTP example

Let's take a look at the "heart" of our firewall and the new rule that allows incoming HTTP

connections:

```
iptables -P INPUT DROP
iptables -A INPUT -i ! ${UPLINK} -j ACCEPT
iptables -A INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT
#our new rule follows
iptables -A INPUT -p tcp --dport http -m state --state NEW -j ACCEPT
iptables -A INPUT -p tcp -i ${UPLINK} -j REJECT --reject-with tcp-reset
iptables -A INPUT -p udp -i ${UPLINK} -j REJECT --reject-with icmp-port-unreachab
```

This new rule allows incoming NEW TCP packets destined for our machine's port 80 (http) to come in. Notice the placement of this rule. It's important that it appears before our REJECT rules. Since iptables will apply the first matching rule, putting it after our REJECT lines would cause this rule to have no effect.

Our final firewall script

Now, let's take a look at our final firewall script, one that can be used on a laptop, workstation, router, or server (or some combination thereof!).

```
#!/bin/bash

#Our complete stateful firewall script.  This firewall can be customized for
#a laptop, workstation, router or even a server. :)

#change this to the name of the interface that provides your "uplink"
#(connection to the Internet)

UPLINK="eth1"

#if you're a router (and thus should forward IP packets between interfaces),
#you want ROUTER="yes"; otherwise, ROUTER="no"

ROUTER="yes"

#change this next line to the static IP of your uplink interface for static SNAT,
#"dynamic" if you have a dynamic IP.  If you don't need any NAT, set NAT to "" to
#disable it.

NAT="1.2.3.4"

#change this next line so it lists all your network interfaces, including lo

INTERFACES="lo eth0 eth1"

#change this line so that it lists the assigned numbers or symbolic names (from
#/etc/services) of all the services that you'd like to provide to the general
#public.  If you don't want any services enabled, set it to ""

SERVICES="http ftp smtp ssh rsync"
```

```
if [ "$1" = "start" ]
then
        echo "Starting firewall..."
        iptables -P INPUT DROP
        iptables -A INPUT -i ! ${UPLINK} -j ACCEPT
        iptables -A INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT

        #enable public access to certain services
        for x in ${SERVICES}
        do
                iptables -A INPUT -p tcp --dport ${x} -m state --state NEW -j AC
        done

        iptables -A INPUT -p tcp -i ${UPLINK} -j REJECT --reject-with tcp-reset
        iptables -A INPUT -p udp -i ${UPLINK} -j REJECT --reject-with icmp-port-u

        #explicitly disable ECN
        if [ -e /proc/sys/net/ipv4/tcp_ecn ]
        then
                echo 0 > /proc/sys/net/ipv4/tcp_ecn
        fi

        #disable spoofing on all interfaces
        for x in ${INTERFACES}
        do
                echo 1 > /proc/sys/net/ipv4/conf/${x}/rp_filter
        done

        if [ "$ROUTER" = "yes" ]
        then
                #we're a router of some kind, enable IP forwarding
                echo 1 > /proc/sys/net/ipv4/ip_forward
                if [ "$NAT" = "dynamic" ]
                then
                        #dynamic IP address, use masquerading
                        echo "Enabling masquerading (dynamic ip)..."
                        iptables -t nat -A POSTROUTING -o ${UPLINK} -j MASQUERADE
                elif [ "$NAT" != "" ]
                then
                        #static IP, use SNAT
                        echo "Enabling SNAT (static ip)..."
                        iptables -t nat -A POSTROUTING -o ${UPLINK} -j SNAT --to
                fi
        fi

elif [ "$1" = "stop" ]
then
        echo "Stopping firewall..."
        iptables -F INPUT
        iptables -P INPUT ACCEPT
        #turn off NAT/masquerading, if any
        iptables -t nat -F POSTROUTING
fi
```

Section 7. Building a better server firewall

Server improvements

It's often possible to make a firewall just an eensy bit "better". Of course, what "better" means depends on your specific needs. Our existing script could meet yours exactly, or maybe some additional tweaking is in order. This section is intended to serve as a cookbook of ideas, demonstrating ways to enhance your existing stateful firewall.

Logging techniques

So far, we haven't discussed how to go about logging anything. There's a special target called LOG that you can use to log things. Along with LOG, there's a special option called "--log-prefix" that allows you to specify some text that will appear alongside the packet dump in the system logs. Here's an example log rule:

```
iptables -A INPUT -j LOG --log-prefix "bad input:"
```

You wouldn't want to add this as the first rule in your INPUT chain, as it would cause a log entry to be recorded for every packet that you receive! Instead, place log rules further down in your INPUT chain with the intention of logging strange packets and other anomalies.

Logging techniques, continued

Here's an important note about the LOG target. Normally, when a rule matches, a packet is either accepted, rejected, or dropped, and no further rules are processed. However, when a log rule matches, the packet is logged. However, it is not accepted, rejected, or dropped. Instead, the packet continues on to the next rule, or the default chain policy is applied if the log rule is the last on the chain.

The LOG target can also be combined with the "limit" module (described in the iptables man page) to minimize duplicate log entries. Here's an example:

```
iptables -A INPUT -m state --state INVALID -m limit --limit 5/minute -j LOG --log
```

Creating your own chains

iptables allows you to create your own user-defined chains that can be specified as targets in your rules. If you want to learn how to do this, spend some time going through Rusty's excellent *http://netfilter.samba.org/unreliable-guides/packet-filtering-HOWTO/index.html* .

Enforcing network usage policy

Firewalls offer a lot of power for those who want to enforce a network usage policy for a corporate or academic LAN. You can control what packets your machine forwards by adding rules to and setting policy for the FORWARD chain. By adding rules to the OUTPUT chain, you can also control what happens to packets that are generated locally, by users on the Linux box itself. iptables also has the incredible ability to filter locally-created packets based on owner (uid or gid). For more information on this, search for "owner" in the iptables man page.

Other security angles

In our example firewall, we've assumed that all internal LAN traffic is trustworthy, and that only incoming Internet traffic must be carefully monitored. Depending on your particular network, that may or may not be the case. There's certainly nothing stopping you from configuring your firewall to provide protection from incoming LAN traffic. Consider other "angles" of your network that you may want to protect. It may also be appropriate to configure two separate LAN security "zones", each with its own security policy.

Section 8. Resources

tcpdump

In this section, I'll point out a number of resources that you'll find helpful as you put together your own stateful firewall. Let's start with an important tool...

tcpdump is an essential tool for exploring low-level packet exchanges and verifying that your firewall is working correctly. If you don't have it, get it. If you've got it, start using it.

---

netfilter.kernelnotes.org

http://netfilter.samba.org is the home page for the netfilter team. There are lots of excellent resources on this page, including the iptables sources, as well as Rusty's excellent *"unreliable guides"* . These include a basic networking concepts HOWTO, a netfilter (iptables) HOWTO, a NAT HOWTO, and a netfilter hacking HOWTO for developers. There's also a *netfilter FAQ* available, as well as other things.

---

iptables man page

Thankfully, there are a lot of good online netfilter resources; however, don't forget the basics. The iptables man page is very detailed and is a shining example of what a man page should be. It's actually an enjoyable read.

Advanced Linux routing and traffic control HOWTO

There's now an *Advanced Linux Routing and Traffic Control HOWTO* available. There's a good section that shows how to use iptables to mark packets, and then use Linux routing functionality to route the packets based on these marks. **Note: This HOWTO contains references to Linux's traffic control (quality of service) functionality (accessed through the new "tc" command). This new functionality, although very cool, is very poorly documented, and attempting to figure out all aspects of Linux traffic control can be a very frustrating task at this point.**

Mailing lists

There's a *netfilter (iptables) mailing list* available, as well as *one for netfilter developers* . You can also access the mailing list archives at these URLs.

Building Internet Firewalls, Second Edition

In June 2000, O'Reilly released an excellent book -- *Building Internet Firewalls, Second Edition* . It's great reference book, especially for those times when you want to configure your firewall to accept (or flat-out reject) a little-known protocol that you're unfamiliar with.

Well, that's it for our resources list, and our tutorial is complete. I hope that this tutorial has been helpful to you, and I look forward to your feedback.

Your feedback

We look forward to getting your feedback on this tutorial. Additionally, you are welcome to contact the author, Daniel Robbins, at *drobbins@gentoo.org* .

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The Toot-O-Matic tool is a short Java program that uses XSLT stylesheets to convert the XML source into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML.