Firewalls and Internet Security

(Mini-Project IV Semester)

IIIT Allahabad - Batch 2000

Documented and Presented By -

Javed Shakeel

Varun Shrivastava

Manish Anilkumar Barman

Project Guided By -

Ranjan Shrivastava

Table of Contents

Acknowledgment

We would like to extend our gratitude and gratefulness to our Project guide, Mr. Ranjan Shrivastava, who helped us on several occasions during the course of this project.
We'd also like to thank our batchmates for the support offered by them during the entire course of the project.

Purpose of this Paper...

The Project undertaken by our group is designed to explore the problem of Internet Security and focus primarily on Firewalls as part of an effective strategy to solve the problem.

This paper presents some of the information acquired about Firewalls during the course of the Project. It also includes a report of the work that went into the configuration of the Firewall.

The First Part (Introduction) addresses the issue of Internet Security, why, where and for whom is it desirable. It also comprises of a short description of preliminary concepts about Firewalls in general, what they are, where and why are they required, how are they implemented and what makes Firewalls do their automated job properly.

The Second Part (The Project: Setting up a Packet Filter Firewall on a Linux Machine) depicts the actual work done during the Project and the results obtained thereof.

This paper has been compiled and documented to present the information collected by our group, which, we hope, will be of assistance to others trying to setup a firewall for their own needs.

Introduction

Considering the popularity and the reach of the Internet into the personal and professional lives of people today, it becomes evident that, like real life, Privacy and Protection need to be taken care of in the cyber world, too.
Firewalls are part of the gamut of tools available to assist in achieving Internet Security for a Network.

What are Firewalls? (The Basics)

A Firewall is basically a protective device. When one connects to the Internet, there are three things that are put to risk

- Local Data
  - Secrecy
  - Integrity
  - Availability to Self
- Computer Resources
- One's Reputation

Firewalls are essentially designed to protect against attacks that hamper local resources including data, not the communication over the Internet. They act as a barrier between the internal network and the Internet so as to screen through only the information that is considered safe by the Network Administrator.

In theory, a Firewall serves multiple purposes

- It restricts entering into the Network at carefully controlled points
- It prevents attacks from getting close to interior defenses
- It restricts leaving the Network at carefully controlled points

Logically, a Firewall is a separator, restrictor and an analyzer.

Where are Firewalls needed and Why?

Firewalls are one of the most indispensable components of the System for Security Conscious Users, Networks offering Services to a group of users or the World Wide Web, Corporations involved in e-commerce as well as Educational Institutions.

This is because any machine connected to the Internet is subject to several **attacks**. Some of them can be enumerated as
Intrusion: Penetrating into the Local System to utilize resources pretending to be a legitimate user. This is the most common attack on a machine connected to the Internet.

Denial of Service: This class of attacks is aimed purely at disrupting the services offered by the machine so that the users of the service are unable to use it at all.

Information Theft: This type of attack deals with compromising the Secrecy of Data by simply acquiring a copy of the information serviced by a computer with the only difference that the information is handed over into the wrong hands.

Firewalls are primarily needed to prevent, or at least, rarify the occurrence of such attacks. Besides, if and when they do occur, the firewall is meant to help in tracing down the origin of the crime.

Types of Firewall Implementations

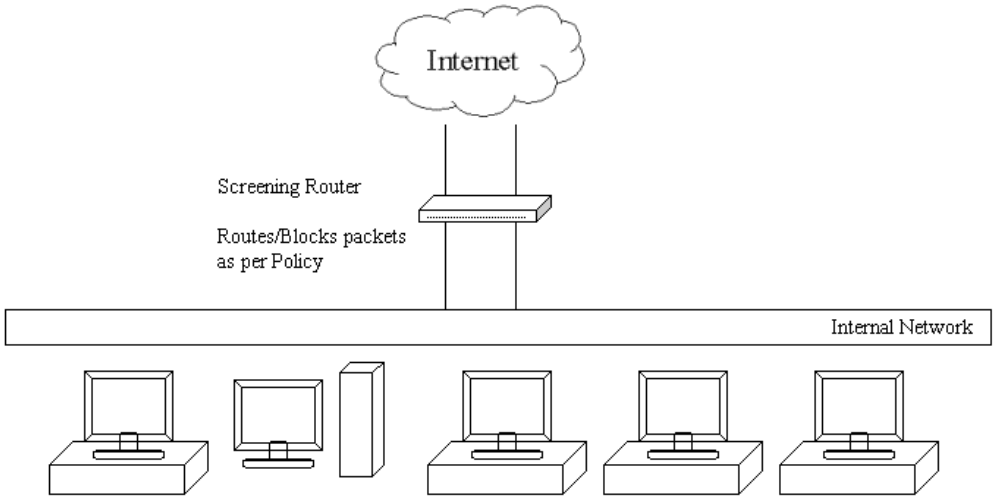
A Firewall is usually installed at the point where the protected Internal Network connects to the Internet. Thus, it gets a chance to screen all the incoming and outgoing traffic thereby allowing only acceptable traffic to pass through and deal with other traffic as specified in the security policy.

The physical implementation of a Firewall is most often, a combination of a set of Hardware components (Routers) and appropriate software.

Firewall Technologies

Packet Filtering (Network Layer)

A filtering firewall works at the network level. Data is only allowed to leave the system if the firewall rules allow it. As packets arrive they are filtered by their type, source address, destination address, and port information contained in each packet.



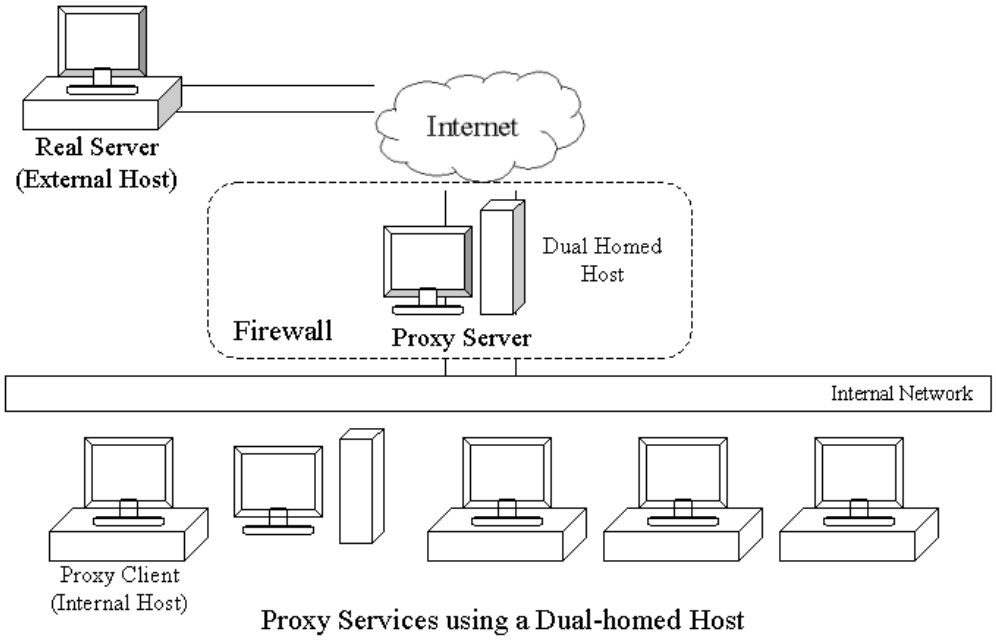
A Screening Router to do packet Filtering

Because very little data is analyzed and logged, filtering firewalls take less CPU and create less latency in your network.

Filtering firewalls do not provide for password controls. Users can't identify themselves. The only identity a user has is the IP number assigned to their workstation.

Proxy Services (Application Layer)

Proxies are mostly used to control, or monitor, outbound traffic. Some application proxies cache the requested data. This lowers bandwidth requirements and decreases the access time for the same data for the next user. It also gives unquestionable evidence of what was transferred.



Proxy Services using a Dual-homed Host

There are two types of proxy servers.

1. Application Proxies - that do the work for the application (user).
2. SOCKS Proxies - that cross wire ports to establish connections.

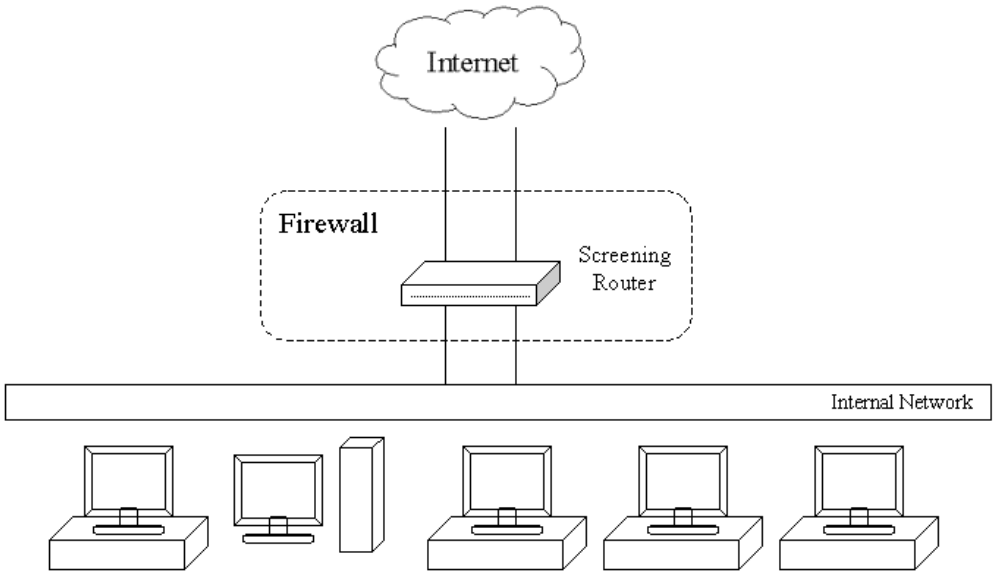
Besides the technology, the configuration of a firewall server requires planning and designing an efficient architecture to implement the Firewall.

Typical Architectures for Firewall Implementation

Single-Box Architecture

These are the simplest Firewall Architectures and have a single object that acts as a Firewall.
The only advantage they have to offer is that they're easy to implement, test and maintain.
They do not offer defense in depth and hence are not very secure. All the security is concentrated at a single point. If that fails, the entire security framework collapses.

Screening Routers and Dual-Homed Hosts are classic examples of Single Box Architectures. These are low cost implementations of Firewalls on a network.



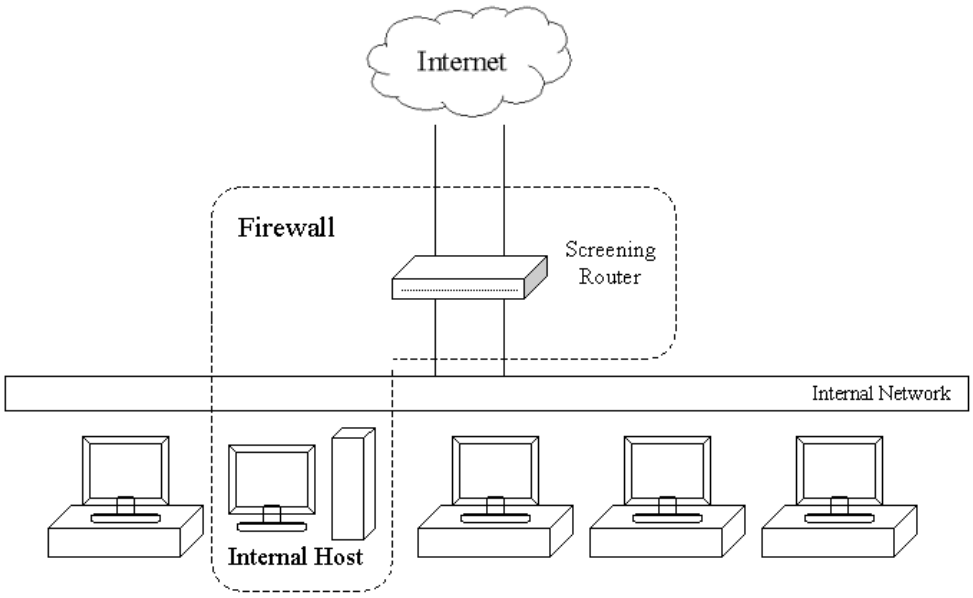
Single Box Architecture

The most appropriate places to setup such architectures are:

- When the Network to be protected is small
- No Services are being provided to the Internet

Screened Host Architecture

Such an architecture comprises of a router configured to permit or deny traffic based on a set of permission rules installed by the administrator and a host on a network behind the screening router. The degree to which a screened host may be accessed depends on the screening rules in the router. The screened host is connected to the Internal network using a separate router. The primary security is provided by packet filtering.



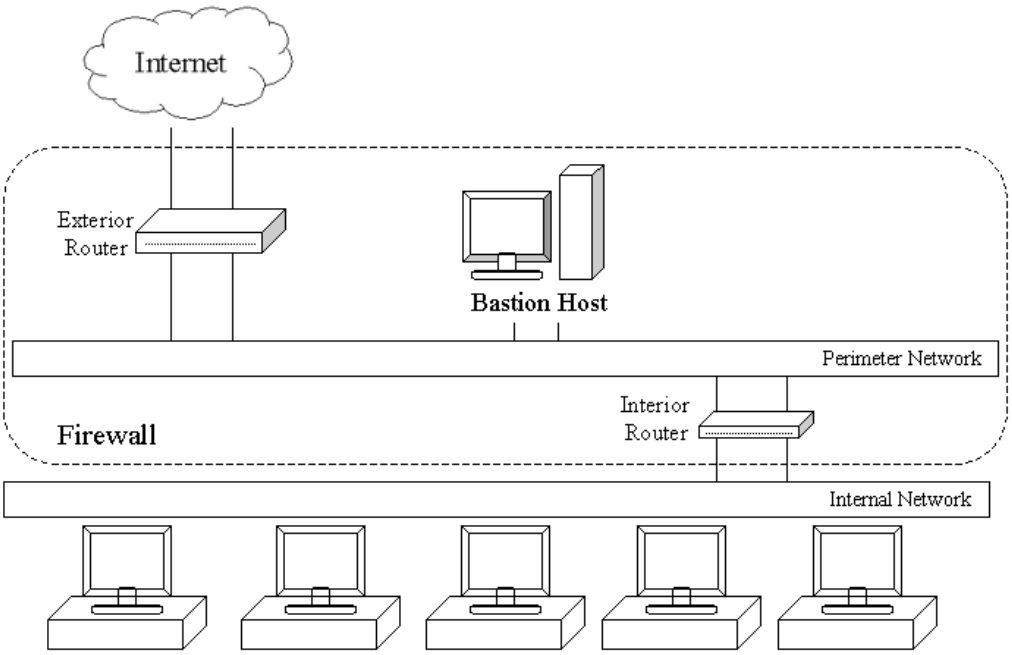
Screened Host Architecture

The most appropriate places to setup such architectures are:

- There are few connections to and from the Internet
- The Network being protected has a high level of host security

Screened Subnet Architecture

This architecture comprises of a subnet behind a screening router. This adds an extra layer of security to the Screened host architecture that provides defense in depth. Breaking into the host doesn't make the internal hosts completely vulnerable. The degree to which the subnet may be accessed depends on the screening rules in the router.



Screened Subnet Architecture

This and multiple variants of this architecture are suitable for most uses.

Security Strategies implemented using Firewalls

After satisfying the hardware requirements for the firewall desired, some configurations need to be put into place to make the firewall do its work. A firewall is able to screen out communication based on the policy defined by the Network Administrator. There are two basic approaches/stances that can be taken depending on the needs of the network..

Default Deny

Prohibit all communication that is not expressly permitted. This kind of stance makes sense from a security point of view and is an obvious choice for administrators. However, the implications of such a strategy is not very helpful to the users of the network.

Default Permit

Permit all communication that is not explicitly prohibited. This stance is more appealing to users but is extremely risky and takes into consideration only things that the Network Administrator can predict beforehand to be capable of compromising the security of the network.

Besides these, there are several simple strategies that can be employed to enhance the security of the network using Firewalls:

Least Privilege

This involves designing operational aspects of a system to operate with a minimum amount of system privilege. This reduces the authorization level at which various actions are performed and decreases the chance that a process or user with high privileges may be caused to perform unauthorized activity resulting in a security breach.

Defense in Depth

It is a security approach whereby each system on the network is secured to the greatest possible degree. This approach is usually used in conjunction with firewalls.

Choke Point

A Choke Point forces attackers to use a narrow channel to penetrate the network allowing for easy monitoring and screening of traffic entering and leaving the network.

Shortcomings of Firewalls

Like all technologies, a firewall has its share of pros and cons. A few of the shortcomings that a network having a firewall at its gateway is subject to are:

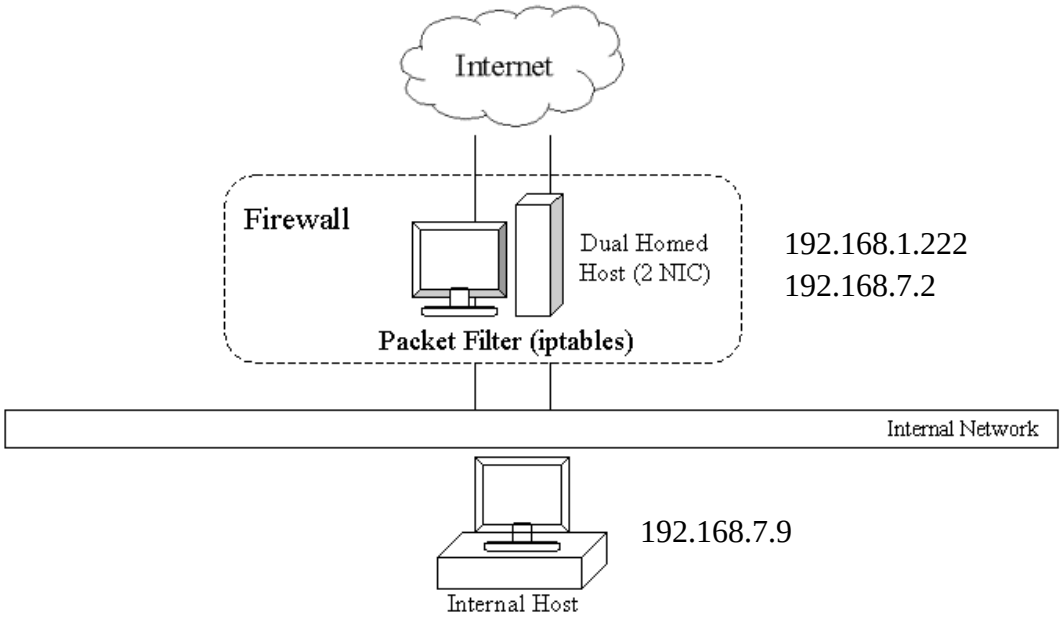
- It can't protect against malicious insiders
- It can't protect against connections that don't go through it
- It can't protect against completely new threats
- It can't fully protect against viruses

Firewalls encompass a variety of technologies...in fact, anything that is designed to protect an internal system from external threats is a Firewall. All approaches and implementations are beyond the scope of this document.
However, a detailed explorative study has been conducted into one of the approaches used to build Firewalls and is described in the next Section.

The Project

Setting up a Packet Filter Firewall on a Linux Machine

A brief description of the configuration setup during the course of the project:
The Firewall designed and implemented uses a Packet Filter (Network Layer) Technology and is implemented in a Single Box Architecture using a Dual-Homed Host as the router and a dummy host representing the internal network.



Packet Filter using a Dual-homed Host

There were three major steps in the process of setting up the entire System:

Setting up an Internal Network
A Dual-Homed Host configured using iproute2 (for IP Forwarding and Masquerading)

A dual homed gateway is a system that has two or more network interfaces, each of which is connected to a different network. In firewall configurations, a dual homed gateway usually acts to block or filter some or all of the traffic trying to pass between the networks.

The iproute2 Package was built into the Linux Kernel and the host was configured to allow for IP Routing and Forwarding. This host acted as a gateway for the internal network (The Host with the IP 192.168.7.9)
This setup was a preliminary requirement to the configuration of the Firewall so as to ensure passage of all traffic to the internal host through a choke point.

After the entire setup was ready and the internal host was able to connect to external hosts through the gateway, the firewall was to be put into place.

Setting up Rules (Network Security Policy) using iptables (A Packet Filter on Linux)

Packet Filtering is the type of firewall implemented as part of the project. The Package iptables was built into the kernel and rules were configured to monitor and filter out packets that went through our gateway.

iptables uses the mechanism of chains to filter out packets passing through the host. There are three default chains **input, output and forward** that determine the fate of the packets destined for the host, generated from the host and passing through the host respectively.
Rules such as accept, deny, reject, log, pass to another chain, etc. can be setup on all of the chains based on various fields in the packet header such as source and destination addresses, protocols, options, etc.
Besides the default chains provided, user defined chains can be created and packets can be passed from chain to chain based on simple if else conditions.

A detailed description of iptables and some of the sample policies experimented with along with their setup procedures are discussed at a later stage is the *Details* section.

Besides configuring using handfed commands, we also designed a file format to save predefined policies in configuration files and an executable to implement a security policy defined in the configuration file.
The File Format is described in the section *Firewall.conf*
You may also find the source code of the executable and a sample configuration file in the appendices.

Advanced Routing and Bandwidth Allocation using iproute2 and tc (Traffic Controller)

A third part of the project was advanced routing and bandwidth control to allow for fair queuing of packets from the hosts on the internal network.
In conjunction with iproute2, a package called tc was used to allocate definite bandwidths to the hosts on the internal Network.

A detailed description of the configuration procedures is discussed at a later point.

Details

Netfilter/Iptables

What is Netfilter/Iptables?

The netfilter/iptables project is the Linux 2.4.x / 2.5.x firewalling subsystem. It delivers you the functionality of packet filtering (stateless or stateful), all different kinds of NAT (Network Address Translation) and packet mangling. It can be used for for all kinds of firewalling, NAT or other advanced packet processing. The major part of netfilter/iptables (doing all the hard work) is included in the standard Linux Kernel. In order to do your runtime configuration of the firewalling subsystem, you will need the *iptables* userspace command.

It is the re-designed and heavily improved successor of the previous 2.2.x ipchains and 2.0.x ipfwadm systems. netfilter is a set of hooks inside the linux 2.4.x kernel's network stack which allows kernel modules to register callback functions called every time a network packet traverses one of those hooks. iptables is a generic table structure for the definition of rulesets. Each rule within an IP table consists out of a number of classifiers (matches) and one connected action (target). netfilter, iptables and the connection tracking as well as the NAT subsystems together build the whole framework.

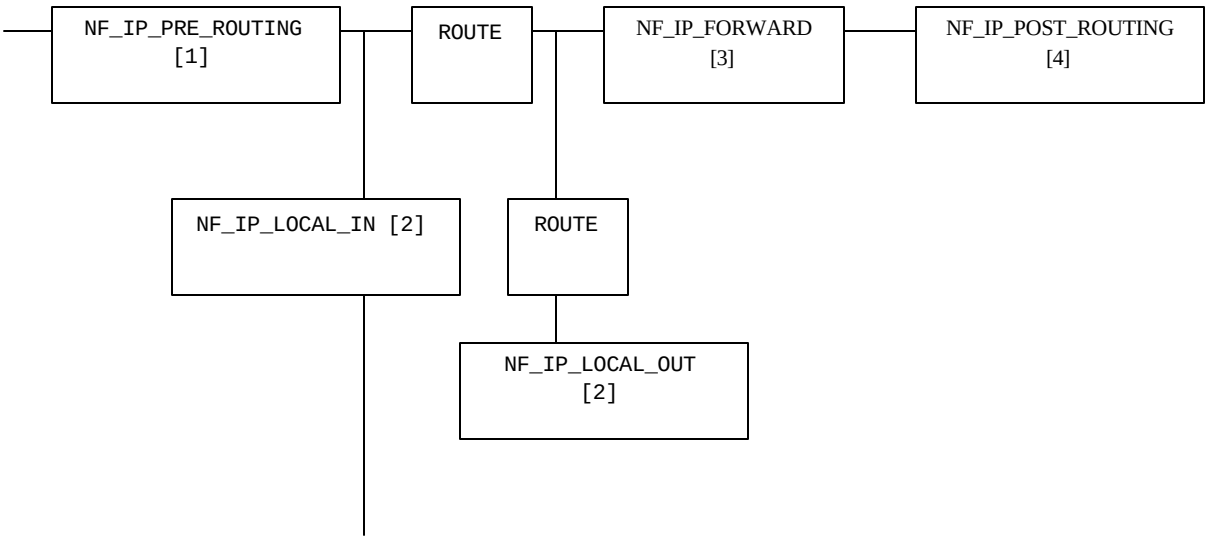
Main Features

- stateful packet filtering (connection tracking)
- all kinds of network address translation
- flexible and extensible infrastructure
- large number of additional features as patches

Netfilter Architechture

Netfilter is merely a series of hooks in various points in a protocol stack (at this stage, IPv4, IPv6 and DECnet). The (idealized) IPv4 traversal diagram looks like the following:

A Packet Traversing the Netfilter System:



On the left is where packets come in: having passed the simple sanity checks (i.e., not truncated, IP checksum OK, not a promiscuous receive), they are passed to the netfilter framework's NF_IP_PRE_ROUTING [1] hook.

Next they enter the routing code, which decides whether the packet is destined for another interface, or a local process. The routing code may drop packets that are unroutable.

If it's destined for the box itself, the netfilter framework is called again for the NF_IP_LOCAL_IN [2] hook, before being passed to the process (if any).

If it's destined to pass to another interface instead, the netfilter framework is called for the NF_IP_FORWARD [3] hook.

The packet then passes a final netfilter hook, the NF_IP_POST_ROUTING [4] hook, before being put on the wire again.

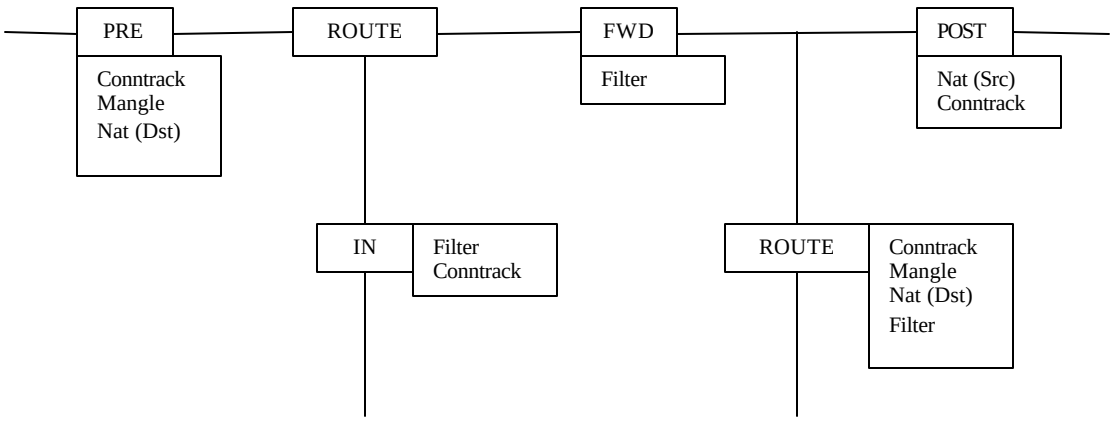
The NF_IP_LOCAL_OUT [5] hook is called for packets that are created locally. Here you can see that routing occurs after this hook is called: in fact, the routing code is called first (to figure out the source IP address and some IP options): if you want to alter the routing, you must alter the `skb->dst' field yourself, as is done in the NAT code.

Packet Selection: IP Tables

A packet selection system called IP Tables has been built over the netfilter framework. It is a direct descendent of ipchains (that came from ipfwadm, that came from BSD's ipfw IIRC), with extensibility. Kernel modules can register a new table, and ask for a packet to traverse a given table. This packet selection method is

used for packet filtering (the `filter' table), Network Address Translation (the `nat' table) and general pre-route packet mangling (the `mangle' table).

The hooks that are registered with netfilter are as follows (with the functions in each hook in the order that they are actually called):



Packet Filtering

This table, `filter', should never alter packets: only filter them.

One of the advantages of iptables filter over ipchains is that it is small and fast, and it hooks into netfilter at the NF_IP_LOCAL_IN, NF_IP_FORWARD and NF_IP_LOCAL_OUT points. This means that for any given packet, there is one (and only one) possible place to filter it. This makes things much simpler for users than ipchains was. Also, the fact that the netfilter framework provides both the input and output interfaces for the NF_IP_FORWARD hook means that many kinds of filtering are far simpler.

Note: I have ported the kernel portions of both ipchains and ipfwadm as modules on top of netfilter, enabling the use of the old ipfwadm and ipchains userspace tools without requiring an upgrade.

NAT

This is the realm of the `nat' table, which is fed packets from two netfilter hooks: for non-local packets, the NF_IP_PRE_ROUTING and NF_IP_POST_ROUTING hooks are perfect for destination and source alterations respectively. If CONFIG_IP_NF_NAT_LOCAL is defined, the hooks NF_IP_LOCAL_OUT and NF_IP_LOCAL_IN are used for altering the destination of local packets.

This table is slightly different from the `filter' table, in that only the first packet of a new connection will traverse the table: the result of this traversal is then applied to all future packets in the same connection.

Masquerading, Port Forwarding, Transparent Proxying
I divide NAT into Source NAT (where the first packet has its source altered), and Destination NAT (the first packet has its destination altered).

Masquerading is a special form of Source NAT: port forwarding and transparent proxying are special forms of Destination NAT. These are now all done using the NAT framework, rather than being independent entities.

Packet Mangling

The packet mangling table (the `mangle' table) is used for actual changing of packet information. It hooks into netfilter at the NF_IP_PRE_ROUTING and NF_IP_LOCAL_OUT points.

Connection Tracking

Connection tracking is fundamental to NAT, but it is implemented as a separate module; this allows an extension to the packet filtering code to simply and cleanly use connection tracking (the `state' module).

How a rule is built

A rule could be described as the pure rules the firewall will follow when blocking different connections and packets in each chain. Each line you write that's inserted to a chain should be considered a rule.

Basics

As we've already explained each rule is a line that the kernel looks at to find out what to do with a packet. If all the criteria's, or matches, are met, we perform the target, or jump, instruction. Normally we'd write a rule something like this:

iptables [table] <command> <match> <target/jump>

There is nothing that says that the target instruction must be last in the line, however, you'd do this normally to get a better readability.

If you want to use another table than the standard table, you could insert the table specification here. However, it is not necessary to specify it explicitly all the time since iptables per default uses the *filter* table to implement your commands on.

The **command** variable tells the iptables command what to do. We use this first variable to tell the program what to do, for example to insert a rule or to add a rule to the end of the chain, or to delete a rule.

The **match** is the part which we send to the kernel that says what a packet must look like to be matched. We could specify what IP address the packet must come from, or from which network interface the packet must come from etc. There are lots of other matches which we can specify here.

Finally we have the **target** of the packet. If all the matches are met for a packet we tell the kernel to perform this action on the packet. We could tell the kernel to send the packet onto another chain that we create ourselves, which must be part of this table. We could tell the kernel to drop this packet dead and do no further processing, or we could tell kernel to send a specified reply to be sent back.

Table

The -t option specifies which table to use. By default, the filter table is used. The following options are available to the -t command:

1. **nat**
   The nat table is used mainly for Network Address Translation. Packets in a stream only traverse this table once. The first packet of a stream is allowed, we presume. The rest of the packets in the same stream are automatically NAT'ed or Masqueraded etc, in case they are supposed to have those actions taken on them. The rest of the packets in the stream will in other words not go through this table again, but instead they will automatically have the same actions taken to them as the first packet in the stream. This is one reason why you should not do any filtering in this table.

   Chains Inside nat Table

   a) PREROUTING

   The PREROUTING chain is used to alter packets as soon as they get in to the firewall.

   b) OUTPUT

   The OUTPUT chain is used for altering locally generated packets (i.e., on the firewall) before they get to the routing decision.

   c) POSTROUTING

   POSTROUTING chain is used to alter packets just as they are about to leave the firewall.

2. **filter**

The filter table should be used for filtering packets generally. For example, we could DROP, LOG, ACCEPT or REJECT packets in this table. There are three chains built in to this table.

Chains Inside filter Table

a) INPUT

The INPUT chain is used on all packets that are destined for our local host (the firewall).

b) OUTPUT

The OUTPUT chain is used for all locally generated packets.

c) FORWARD

The FORWARD chain is used on all non-locally generated packets that are not destined for our local host (the firewall, in other words).

3. mangle

This table is used mainly for mangling packets. We could change different packets and how their headers look among other things. Examples of this would be to change the TTL, TOS or MARK. Note that the MARK is not really a change to the packet, but a mark for the packet is set in kernel-space which other rules or programs might use further on in the firewall to filter or do advanced routing on with ***tc*** as an example.

Chains Inside mangle Table

a) PREROUTING

PREROUTING is used for altering packets just as they enter the firewall and before they hit the routing decision.

b) OUTPUT

OUTPUT is used for changing and altering locally generated packets before they enter the routing decision.

Commands

The command tells iptables what to do with the rest of the commandline that we send to the program.

-A, **--append**

This command appends the rule to the end of the chain. The rule will in other words always be put last in the ruleset in comparison to previously added rules, and hence be checked last unless you append more rules later on.
For e.g.,

```
iptables –A INPUT ...
```

-D, **--delete**

This command deletes a rule in a chain. This could be done in two ways, either by specifying a rule to match with the -D option (as in the first example) or by specifying the rule number that we want to match. If you use the first way of deleting rules, they must match totally to the entry in the chain. If you use the second way, the rules start getting numbered at the beginning of each chain, and the top rule is number 1.
For e.g.,

```
iptables -D INPUT –dport 80 -j DROP
iptables -D INPUT 1
```

-L, **--list**

This command lists all the entries in the specified chain. In the above case, we would list all the entries in the INPUT chain. It's also legal to not specify any chain at all. In the last case, the command would list all the chains in the specified table (To specify table, see section Traversing of tables and chains). The exact output is affected by other options sent to the program, for example -n -v etc. For e.g.,

```
iptables -t filter -L
```

-F, **--flush**

This command flushes the specified chain from all rules and is equivalent to deleting each rule one by one but is quite a bit faster. The command can be used without options, and will then delete all rules in all chains within the specified table. For e.g.,

```
iptables -F INPUT
```

-N, **--new-chain**

This command tells the kernel to create a new chain by the specified name in the specified table. In the example below, we create a chain called allowed. Note that there must be no target of the same name as the new chain. For e.g.,

```
iptables -N allowed
```

-X, –delete-chain

This command deletes the specified chain from the table. For this command to work, there must be no rules that are referring to the chain that's being deleted. In other words, you'd have to replace or delete all rules referring to the chain before actually deleting the chain. If this option is used without any options, all non-built-in chains are deleted from the specified table. For e.g.,

```
iptables -X allowed
```

-P, –policy

This command tells the kernel to set a specified default target, or policy, on a chain. All packets that don't match any rule will then be forced to use the policy of the chain. Note that user-defined chains cannot have policies. For e.g.,

```
iptables -P INPUT DROP
```

Matches

The matches can be broken into five different subcategories. First of all there are generic matches which are generic and can be used in all rules. Then we have the TCP matches which can only be applied to TCP packets. We have UDP matches which can only be applied to UDP packets and ICMP matches which can only be used on ICMP packets. Finally we have special matches such as the state, owner and limit matches and so on.

Generic Matches

A generic match is a kind of match that is always available whatever kind of protocol we are working on or whatever match extensions we have loaded. No special parameters are in other words needed to load these matches at all. These matches are always available.

-p, --protocol

This match is used to check for certain protocols. Examples of protocols are TCP, UDP and ICMP. This list can vary a bit at the same time since it uses the /etc/protocols, if it can't recognize the protocol itself. First of all the protocol match can take one of the three aforementioned protocols, as well as ALL, which means to match all of the previous protocols. The protocol may also take a numeric value, such as 255 which would mean the RAW IP protocol. Finally, the program knows about all the protocols in the /etc/protocols file as we already explained. The command may also take a comma delimited list of protocols, such as udp, tcp which would match all UDP and TCP packets. If this match is given the numeric value of zero (0), it means ALL protocols, which in turn is the default behavior in case the –protocol match is not used. This match can also be inversed with the ! sign,

so --protocol ! tcp would mean to match the ICMP and UDP protocols. For e.g.,

```
iptables -A INPUT -p tcp
```

-s, --src, --source

This is the source match which is used to match packets based on their source IP address. The main form can be used to match single IP addresses such as 192.168.1.1. It could be used with a netmask in a bits form. One way is to do it with an regular netmask in the 255.255.255.255 form (ie, 192.168.0.0/255.255.255.0), and the other way is to only specify the number of ones (1's) on the left side of the network mask. This means that we could for example add /24 to use a 255.255.255.0 netmask. We could then match whole IP ranges, such as our local networks or network segments behind the firewall. The line would then look something like, for example, 192.168.0.0/24. This would match all packets in the 192.168.0.x range. We could also inverse the match with an ! just as before. If we would in other words use a match in the form of –source ! 192.168.0.0/24 we would match all packets with a source address not coming from within the 192.168.0.x range. The default is to match all IP addresses. For e.g.,

```
iptables -A INPUT -s 192.168.1.1
```

-d, –dst, –destination

This is the match used to match packets based on their destination address or addresses. It works pretty much the same as the –source match and has the same syntax, except that it matches based on where the packets came from. To match an IP range, we can add a netmask either in the exact netmask form, or in the number of ones (1's) counted from the left side of the netmask bits. It would then look like either 192.168.0.0/255.255.255.0 or like 192.168.0.0/24 and both would be equivalent to each-other. We could also invert the whole match with an ! sign, just as before. –destination ! 192.168.0.1 would in other words match all packets except those not destined to the 192.168.0.1 IP address.

```
iptables -A INPUT -d 192.168.1.1
```

-i, –in-interface

This match is used to match based on which interface the packet came in. Note that this option is only legal in the INPUT, FORWARD and PREROUTING chains and will render an error message when used anywhere else. The default behavior of this match, in case the match isn't specified is to assume a string value of "+". The "+" value is used to match a string of letters and numbers. A single "+" would in other words tell the kernel to match all packets without considering which interface it came in on. The + string can also be used at the end of an interface, and eth+ would in other words match all ethernet devices. We can also invert the meaning of

this option with the help of the ! sign. The line would then have a syntax looking something like -i ! eth0, which would mean to match all incoming interfaces, except eth0. For e.g.,

```
iptables -A INPUT -i eth0
```

-o, –out-interface

The –out-interface match is used to match packets depending on which interface it's going out on. Note that this match is only available in the OUTPUT, FORWARD and POSTROUTING chains, in opposite of the --in-interface match. Other than this, it works pretty much the same as the --in-interface match. The "+" extension is understood so you can match all eth devices with eth+ and so on. To inverse the meaning of the match, you can use the ! sign in exactly the same sense as in the --in-interface match. Of course, the default behavior if this match is left out is to match all devices, regardless of where the packet is going. For e.g.,

```
iptables -A FORWARD -o eth0
```

-f, –fragment

This match is used to match the second and third part of a fragmented packet. The reason for this is that in the case of fragmented packets, there is no way to tell the source or destination ports of the fragments, nor ICMP types, among other things. Also, fragmented packets might in rather special cases be used to compile attacks against computers. Such fragments of packets will not be matched by other rules when they look like this, and hence this match. This option can also be used in conjunction with the ! sign, however, in this case the ! sign must precede the match, like this "! -f". When this match is inversed, we match all head fragments and/or de-fragmented packets. What this means is that we match all the first fragments of a fragmented packets, and not the second, third, and so on, fragments. We also match all packets that has not been fragmented during the transfer. Also note that there are de-fragmentation options within the kernel that can be used which are really good.

```
iptables -A INPUT -f
```

Implicit Matches

Implicit matches are loaded automatically when we tell iptables that this rule will match for example TCP packets with the --protocol match. There are currently 3 types of implicit matches that are loaded automatically for 3 different protocols. These are TCP matches, UDP matches and ICMP matches. The TCP based matches contain a set of different matches that are available for only TCP packets, and UDP based matches contain another set of matches that are available only for UDP packets, and the same thing for ICMP packets. There are also explicitly loaded matches that you must load

explicitly with the -m or –match option which we will go through later on in the next section.

TCP Matches

These matches are protocol specific and are only available when working with TCP packets and streams. To use these matches you need to specify --protocol tcp on the command line before trying to use these matches. Note that the –protocol tcp match must be to the left of the protocol specific matches.

It includes matches for specifying source-port, destination-port, tcp-flags, tcp-options, etc. More information can be obtained from the man page of iptables.

```
iptables –p tcp -h
```

UDP Matches

These matches are implicitly loaded when you specify the –protocol UDP match and will be available after this specification. Note that UDP packets are not connection oriented, and hence there is no such thing as different flags to set in the packet to give data on what the datagram is supposed to do, such as open or closing a connection, or if they are just simply supposed to send data. UDP packets do not require any kind of acknowledgement either. If they are lost, they are simply lost (Not taking ICMP error messaging etcetera into account). This means that there is quite a lot less matches to work with on a UDP packet than there is on TCP packets. Note that the state machine will work on all kinds of packets even though UDP or ICMP packets are counted as connectionless protocols. The state machine works pretty much the same on UDP packets as on TCP packets.

It includes matches for specifying source-port and destination-port. More information can be obtained from the man page of iptables.

```
iptables –p udp -h
```

ICMP Matches

The ICMP protocol is mainly used for error reporting and for connection controlling and such features. ICMP is not a protocol subordinated to the IP protocol, but more of a protocol beside the IP protocol that helps handling errors. The header of an ICMP packet is very similar to that of an IP header, but contains differences. The main feature of this protocol is the type header which tells us what the packet is to do.

It includes match for specifying the icmp-type of the message. ICMP types can be specified either by their numeric values or by their names. Numerical values are specified in RFC 792. To find a complete listing of the ICMP name values, do

```
iptables -p icmp –h
```

This match can also be inverted with the ! sign like

```
iptables -p icmp –icmp-type ! 8
```

Explicit Matches

Explicit matches are matches that must be specifically loaded with the -m or --match option. The difference between implicitly loaded matches and explicitly loaded ones is that the implicitly loaded matches will automatically be loaded when you, for example, match TCP packets, while explicitly loaded matches will not be loaded automatically in any case and it is up to you to activate them before using them.

Explicit matches include STATE match to match packets according to the connection tracking code, MAC match to support matching packets based on there MAC address, LIMIT match to limit the number of times a particular chain is used, MULTIPORT match, MARK match to match packets based on their marks set in the kernel, OWNER match, UNCLEAN match, TOS match to match packets based on the *tos* field in the IP-header, and TTL match to match packets based on the *ttl* field in the IP-header.

Targets/Jumps

The target/jumps tell the rule what to do with a packet that is a perfect match with the match section of the rule. For e.g., ACCEPT, DROP. Targets/jumps are specified by a -j option in a chain. Now let us have a brief look at how a jump is done.
The jump specification is done exactly the same as the target definition except that it requires a chain within the same table to jump to. To jump to a specific chain, it is required that the chain has already been created. For example, let's say we create a chain in the filter table called tcp_packets like this: iptables –N tcp_packets. We could then add a jump target to it like this:
iptables -A INPUT -p tcp -j tcp_packets. We would then jump from the INPUT chain to the tcp_packets chain and start traversing that chain. When/If we reach the end of that chain, we get dropped back to the INPUT chain and the packet starts traversing from the rule one step below where it jumped to the other chain (tcp_packets in this case). If a packet is ACCEPT'ed within one of the subchains, it will automatically be ACCEPT'ed in the superset chain also and it will not traverse any of the superset chains any further. However, do note that the packet will traverse all other chains in the other tables in a normal fashion.

Targets on the other hand specify an action to take on the packet in question. We could for example, DROP or ACCEPT the packet depending on what we want to do. Targets may also end with different results one could say, some targets will make the packet stop traversing the specific chain and superset chains as described above. Good examples of such rules are DROP and ACCEPT. Rules that are stopped will not pass through any of the rules further on in the chain or superset chains. Other targets, may take an action on the packet and then the packet will continue passing through the rest of the rules anyway, a good example of this would be the LOG, DNAT and SNAT targets. These packets may be logged, Network Address Translated and then be passed

on to the other rules in the same chains. This may be good in cases where we want to take two actions on the same packet, such as both mangling the TTL and the TOS value of a specific packet/stream. Some targets will also take options that may be necessary (What address to do NAT to, what TOS to use etcetera) while others have options not necessary, but available in any case (log prefixes, masquerade to ports and so on). Let us look into details of these targets.

ACCEPT

This target is used to accept the packet and allow thereby allow the packet to pass-through the firewall.

DROP

This target is used to silently drop a packet from the firewall. For e.g., you would like to drop unwanted packets.

REJECT

This is used to send back an error packet in response to the matched packet: otherwise it is equivalent to DROP. This target is only valid in the INPUT, FORWARD and OUTPUT chains, and user-defined chains which are only called from those chains.

There is an option to set the type of return packet. This can be set by --reject-with option. It can take the following values icmp-net-unreachable, icmp-host-unreachable, icmp-port-unreachable, icmp-proto-unreachable, icmp-net-prohibitedor icmp-host-prohibited, which return the appropriate ICMP error message (port-unreachable is the default). The option echo-reply is also allowed; it can only be used for rules which specify an ICMP ping packet, and generates a ping reply. Finally, the option *tcp-reset* can be used on rules which only match the TCP protocol.

MIRROR

This is an experimental demonstration target which inverts the source and destination fields in the IP header and retransmits the packet. It is only valid in the INPUT, FORWARD and PREROUTING chains, and user-defined chains which are only called from those chains. Note that the outgoing packets are NOT seen by any packet filtering chains, connection tracking or NAT, to avoid loops and other problems.

SNAT

This target is only valid in the *nat* table, in the POSTROUTING chain. It specifies that the source address of the packet should be modified (and all

future packets in this connection will also be mangled), and rules should cease being examined. It takes one option to specify the IP address which will be used to replace the source IP address in the packet.

DNAT

This target is only valid in the *nat* table, in the PREROUTING and OUTPUT chains, and user-defined chains which are only called from those chains. It specifies that the destination address of the packet should be modified (and all future packets in this connection will also be mangled), and rules should cease being examined. It takes one option to specify the IP address which will be used to replace the destination IP address in the packet.

MASQUERADE

This target is only valid in the *nat* table, in the POSTROUTING chain. It should only be used with dynamically assigned IP (dialup) connections: if you have a static IP address, you should use the SNAT target. Masquerading is equivalent to specifying a mapping to the IP address of the interface the packet is going out. It has the option of specifying a range of source ports to use, overriding the default SNAT source port-selection heuristics. This option is only valid if the rule also specifies *-p tcp* or *-p udp*.

REDIRECT

This target is only valid in the nat table, in the PREROUTING and OUTPUT chains, and user-defined chains which are only called from those chains. It alters the destination IP address to send the packet to the machine itself (locally-generated packets are mapped to the 127.0.0.1address). It has the option that specifies a destination port or range of ports to use without which, the destination port is never altered.

Packet Traversal of Tables and Chains

When a packet first enters the firewall, it hits the hardware and then gets passed on to the proper device driver in the kernel. Then the packet starts to go through a series of steps in the kernel before it is either sent to the correct application locally), or forwarded to another host or whatever happens to it.

Forwarded Packets

1) On the wire
2) Arrives on the interface (eth0, etc.)
3) Passes through table mangle, chain PREROUTING
4) Passes through table nat, chain PREROUTING
5) Kernel makes the routing decision using its routing table

6) Passes through table filter, chain FORWARD
7) Passes through table nat, chain POSTROUTING
8) Packet comes on the outgoing interface
9) On the wire, again

Packets Destined for Local-host

1) On the wire
2) Arrives on the interface (eth0, etc.)
3) Passes through table mangle, chain PREROUTING
4) Passes through table nat, chain PREROUTING
5) Kernel makes the routing decision using its routing table
6) Passes through table filter, chain INPUT
7) Packet gets to the Local process/application

Packets from Local-host

1) Local process/application
2) Passes through table mangle, chain OUTPUT
3) Passes through table nat, chain OUTPUT
4) Passes through table nat, chain OUTPUT
5) Passes through table filter, chain OUTPUT
6) Kernel makes the routing decision using its routing table
7) Passes through table nat, chain POSTROUTING
8) Goes out on some interface (eth0, etc.)
9) On the wire

Firewall.conf

This is a small configuration file made by us to make the firewall rules more readable and easily retrievable. This file can be used to write the rules of IPTables in a more readable manner. Its format specifies some tags which can be used to specify firewall rules. These tags are then parsed by the program (*fire_conf_read*) to convert these tags to corresponding command-line options to the *iptables* user-space command.

A sample entry in this file would be:

```
#giving access of the internal webserver to the outside
#world

$webserver= 192.168.1.1

action=append_rule
target=ACCEPT
table=filter
chain=FORWARD
protocol=TCP
s_interface=ANY
s_addr=ANY
d_interface=ANY
d_addr=$webserver
s_port=ANY
d_port=80
tcp_flags=ANY
tcp_option=ANY

#allowing the reply TCP packets
action=append_rule
target=ACCEPT
table=filter
chain=FORWARD
protocol=TCP
module=state
state=ESTABLISHED
```

In the above entry in *firewall.conf*,

1) *$webserver* variable is a constant that will be used on line number 14 to specify *d_addr*.
2) Anything that is starting with # is a single line comment.
3) The rest of the file follows a format
   tag=value
   for eg: *target=FORWARD* specifies what will be the target of the rule.
4) In every rule the tag *action* is required.

A sample *firewall.conf* file is given in Appendix-A of this document.

Description of Tags

The tags described here are the tags included in the version 0.3 format of *firewall.conf*.

action

Legal Values

append_rule; delete_rule; replace_rule; insert_rule; flush_chain; zero_counters; create_chain; delete_chain; rename_chain; policy_set

Use

This tag is used to specify what basic action this rule is performing. It can take the following values

- **append_rule**
  To append a rule to a chain specified by the tag *chain* in the table specified by the tag *table*.
- **delete_rule**
  To delete a rule from a chain specified by the tag *chain* in the table specified by the tag *table*.
- **replace_rule**
  To replace an existing rule from a chain specified by the tag *chain* in the table specified by the tag *table*.
- **insert_rule**
  To insert a rule in a chain specified by the tag *chain* in the table specified by the tag *table*.
- **flush_chain**
  To flush a chain specified by the tag *chain* in the table specified by the tag *table*.
- **zero_counters**
  To reset the byte counters of a chain specified by the tag *chain* in the table specified by the tag *table*.
- **create_chain**
  To create a new chain of the name specified by the tag *chain* in the table specified by tag *table*.
- **rename_chain**
  To rename a user-defined chain to a new name specified by the tag *chain* in the table specified by the tag *table*.
- **policy_set**
  To set the policy of a built-in chain specified by the tag *chain* in the table specified by the tag *table*.

target

Legal Values

ACCEPT; DROP; REJECT; <any other legal target as specified by IPTables>

Use

Its work is just to provide the IPTables with the *-j* (jump to target) option. Any value that you specify will be appended to the IPTables rule after *-j*. The error checking will be done by the IPTables package will do the error-checking.

table

Legal Values

filter; nat; mangle

Use

This tag is used to specify the table to use for a given operation.

chain

Legal Values

INPUT; FORWARD; OUTPUT; PREROUTING; POSTROUTING; <any user-defined chain>

Use

This tag is used to specify the chain to use for a given operation. The chain specified should be valid in the table specified by the tag *table*.

protocol

Legal Values

ANY; TCP; UDP; <any other protocol specified in /etc/protocols>

Use

This tag is used to specify the protocol to use in a given rule. The value ANY is like specifying no protocol match to a rule.

s_interface

Legal Values

ANY; eth0; ppp0

Use

This tag is used to specify the source-interface to use in a given rule i.e. to match packets which have come from the interface specified by its value. The value ANY is like specifying no source-interface match.

d_interface

Legal Values

ANY; eth0; ppp0

Use

This tag is used to specify the destination-interface to use in a given rule i.e. to match packets which will go through the interface specified by its value. The value ANY is like specifying no destination-interface match.

s_addr

Legal Values

ANY; 192.168.1.4; 192.168.1.4/255.255.255.0; 192.168.1.5/24

Use

This tag is used to specify the source-address to use in a given rule i.e. to match packets which have come from the address specified by its value. The value ANY is like specifying no source-address match.

d_addr

Legal Values

ANY; 192.168.1.4; 192.168.1.4/255.255.255.0; 192.168.1.5/24

Use

This tag is used to specify the destination-address to use in a given rule i.e. to match packets which are destined for the address specified by its value. The value ANY is like specifying no destination-address match.

s_port

Legal Values

<any legal port number>

Use

This tag is used to specify the source-port to use in a given rule i.e. to match packets which have come from the port specified by its value. The value ANY is like specifying no source-port match.

d_port

Legal Values

<any legal port number>

Use

This tag is used to specify the destination-port to use in a given rule i.e. to match packets which destined for the port specified by its value. The value ANY is like specifying no destination-port match.

module

Legal Values

state; multiport; mac; limit; mark; owner; unclean; tos <any other extension module>

Use

This tag is used to specify the IPTables to load any extension module, like *state, multiport, etc.*

state

Legal Values

ESTABLISHED, RELATED, INVALID, NEW

Use

This tag is used to match a packet based on its state given by the *state machine (ip_conntrack.o*).

mac_source

Legal Values

Any legal mac-address allowed by IPTables.

Use

This tag is used to specify the source mac-address to use in a given rule.

limit_rate

Legal Values

Values should be of the form number[/second; /minute; /hour; /day]

Use

This tag is used to specify the maximum average matching rate of a given rule.

limit_burst

Legal Values

<any non-negative number>; 3; 5

Use

This tag is used to specify the maximum initial number of packets to match for a given rule.

port

Legal Values

<any legal port number>

Use

This tag is used to specify both the source and destination port of a packet, i.e. when both the source-port and destination-port are equal and equal to the tags value.

mark

Legal Values

Value should be of the form <unsigned mark value>[/mask]

Use

This tag is used to specify the mark value of a packet to match. If the mask is given then it is logically ANDed with the mark value before the comparison.

uid_owner

Legal Values

<Any legal userid>

Use

This tag is used to match packets that are created by a process with an effective used-id specified by its value.

gid_owner

Legal Values

<Any legal groupid>

Use

This tag is used to match packets that are created by a process with an effective group-id specified by its value.

pid_owner

Legal Values

<Any legal process-id>

Use

This tag is used to match packets that are created by a process with a process-id specified by its value.

sid_owner

Legal Values

<Any legal session-id>

Use

This tag is used to match packets that are created by a process in the given session group specified by its value.

tos

Legal Values

<Any standard name like ‘Minimize Delay’, etc. or a numeric value>

Use

This tag is used to specify the tos (type of service) value to match in the IP-Header of a packet.

log_level

Legal Values

<any level of logging as specified in syslog.conf>

Use

This tag is used to specify the level of logging to use while logging a packet.

log_prefix

Legal Values

<any string upto 29 characters>

Use

This tag is used to specify the log-prefix that will be used while logging a packet.

log_tcp_sequence

Legal Values

y or n

Use

This tag is used to enable or disable the logging of tcp-sequence numbers from the TCP-header of a packet.

log_tcp_options

Legal Values

y or n

Use

This tag is used to enable or disable the logging of tcp-options from the TCP-header of a packet.

log_ip_options

Legal Values

y or n

Use

This tag is used to enable or disable the logging of ip-options from the IP-header of a packet.

set_mark

Legal Values

<any unsigned integer value>

Use

This tag is used to specify the netfilter mark value that will be associated with a packet.

reject_with

Legal Values

icmp-net-unreachable, icmp-host-unreachable, icmp-port-unreachable, icmp-proto-unreachable, icmp-net-prohibitedor, icmp-host-prohibited, echo-reply, tcp-reset

Use

This tag is used to specify the type of packet that will be sent back to the source of a packet while REJECTing a packet.
It can take the following values

- **icmp-net-unreachable**
  To send an ICMP-net-unreachable packet back to the source of the packet.
- **icmp-host-unreachable**
  To send an ICMP-host-unreachable packet back to the source of the packet.
- **icmp-port-unreachable**
  To send an ICMP-port-unreachable packet back to the source of the packet.
- **icmp-proto-unreachable**
  To send an ICMP-proto-unreachable packet back to the source of the packet.
- **icmp-net-prohibitedor**
  To send an ICMP-net-prohibitedor packet back to the source of the packet.
- **icmp-host-prohibited**
  To send an ICMP-host-prohibited packet back to the source of the packet.
- **echo-reply**
  To send an ICMP-echo-reply packet back to the source of the packet. It can only be used in a rule that specify an ICMP ping packet.
- **tcp-reset**
  To send a TCP-RST packet back to the source of the packet. It can only be used in a rule that matches TCP packets.

set_tos

Legal Values

<any legal tos values(number or a name)>

Use

This tag is used to specify the value of tos (Type of Service) that will be set in the IP-header of a packet.

set_ttl

Legal Values

<any legal ttl value (i.e. 0-255)>

Use

This tag is used to specify the value of ttl (Time to Live) that will be set in the IP-header of a packet.

ttl_decrease

Legal Values

<any unsigned integer>

Use

This tag is used to decrease the TTL field of a packet by a given value.

ttl_increase

Legal Values

<any unsigned integer>

Use

This tag is used to increase the TTL field of a packet by a given value.

ttl

Legal Values

< any legal ttl value (i.e. 0-255)>

Use

This tag is used to match packets who have the specified TTL value in the TTL field of the IP-header of a packet.

icmp_type

Legal Values

<any legal icmp type>

Use

This tag is used to match packets which are the type of icmp packet specified by its value.

Fire_conf_read

This is a small program made in C++ that is used to parse the file *firewall.conf* in order to change it into rules that the IPTables command understands.

For e.g.,

For the following entry in the file *firewall.conf*

```
action=append_rule
target=ACCEPT
table=filter
chain=FORWARD
s_interface=eth0
d_interface=eth1
d_port=80
```

The program *fire_conf_read* would produce output:

```
iptables -j ACCEPT -A FORWARD -t filter -i eth0 -o eth1 \
--destination-port 80
```

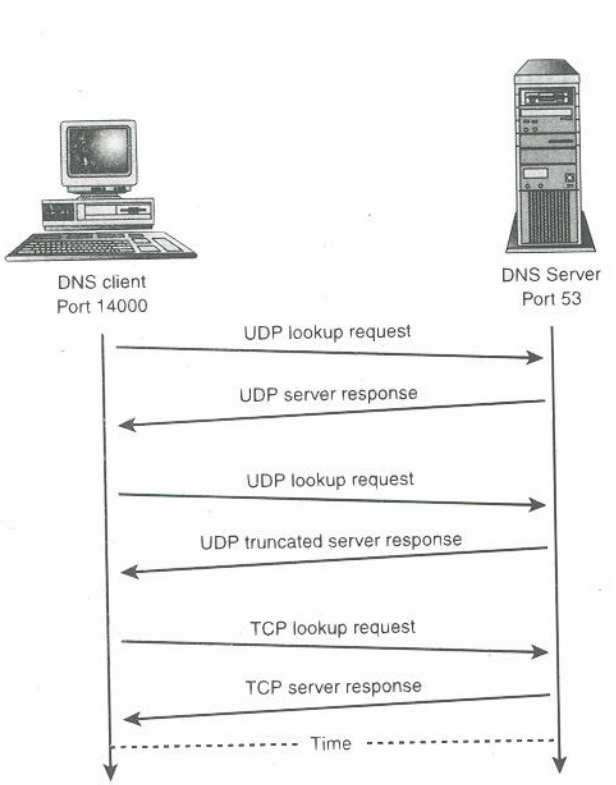
Thus we can store the IPTables rules in a convenient and readable way. The source code of this program is given in Appendix-B of this document.

Enabling Basic, Required Internet Services

1) Allowing DNS (UDP/TCP Port 53)

DNS uses a communication protocol that relies on both UDP and TCP. Connection modes include regular client-to-server connections, peer-to-peer traffic between forwarding servers and full servers, and primary and secondary name servers connections.

Query lookup requests are normally done over UDP, both for client-to-server lookups and peer-to-peer server lookups. The UDP communication can fail for a client-to-server lookup if the information returned is too large to fit in a single UDP DNS packet. The server sets a flag bit in the DNS message header indicating that the data is truncated. In this case, the protocol allows for a retry over TCP. The figure below shows the relationship between UDP and TCP during a DNS lookup. In practice, TCP isn't normally needed for queries. TCP is conventionally needed for administrative zone transfers between the primary and secondary name servers.



DNS client-to-server lookup.

Zone transfers are the transfer of a name server's complete information of a network or a piece (zone) of a network, the server is authoritative for (i.e., the official server). The authoritative name server is referred to as the primary name server. Secondary, or backup, name servers periodically request zone transfers from their primary to keep this DNS caches up-to-date.

DNS Protocol

Description	Protocol	Remote Address	Remote Port	In/Out	Local Address	Local Port	TCP flag
Local client query	UDP	NAMESERVER	53	Out	IPADDR	1024:65535	-
Remote Server response	UDP	NAMESERVER	53	In	IPADDR	1024:65535	-
Local client query	TCP	NAMESERVER	53	Out	IPADDR	1024:65535	Any
Remote Server response	TCP	NAMESERVER	53	In	IPADDR	1024:65535	Ack
Local server query	UDP	NAMESERVER	53	Out	IPADDR	53	-
Remote Server response	UDP	NAMESERVER	53	In	IPADDR	53	-
Local zone transfer request	TCP	Primary	53	Out	IPADDR	1024:65535	Any
Remote zone transfer request	TCP	Primary	53	In	IPADDR	1024:65535	Ack
Remote client query	UDP	DNS Client	1024:65535	In	IPADDR	53	-
Local sever response	UDP	DNS Client	1024:65535	Out	IPADDR	53	-
Remote client query	TCP	DNS Client	1024:65535	In	IPADDR	53	Any
Local server response	TCP	DNS Client	1024:65535	Out	IPADDR	53	Ack
Remote client query	UDP	DNS Client	53	In	IPADDR	53	-
Local server response	UDP	DNS Client	53	Out	IPADDR	53	-
Remote zone transfer request	TCP	Secondary	1024:65535	In	IPADDR	53	Any
Local zone transfer request	TCP	Secondary	1024:65535	Out	IPADDR	53	Ack

Allowing DNS Lookups as a Client

The DNS resolver client isn't a specific program. The client is incorporated into the network library code compiled into network programs. When a hostname requires a lookup, the resolver requests a lookup from a *named* server. Many small systems are configured only as a DNS client. The server runs on a remote machine. For a home user, the name server is usually a machine owned by your ISP.

DNS sends lookup request as a UDP datagram:

```
$IPTABLES –A FORWARD –p UDP \
–i $INTERNAL_INTERFACE \
–d $NAMESERVER –dport 53 \
–j ACCEPT

$IPTABLES –A FORWARD –p UDP –m state \
--state ESTABLISHED \
-j ACCEPT
```

If an error occurs because the returned data is too large to fit in a UDP datagram, DNS retries using a TCP connection.

The next two rules are included for rare occasion when DNS lookup won't fit in a DNS UDP datagram. They won't be used in normal day-to-day operations. You could run your systems without problems for months without these TCP rules. Unfortunately, every so often- perhaps once or twice a year- your DNS lookups hang without these rules due to a poorly configured remote DNS server. More typically, these rules are used by a secondary name server requesting a zone transfer from its primary name server:

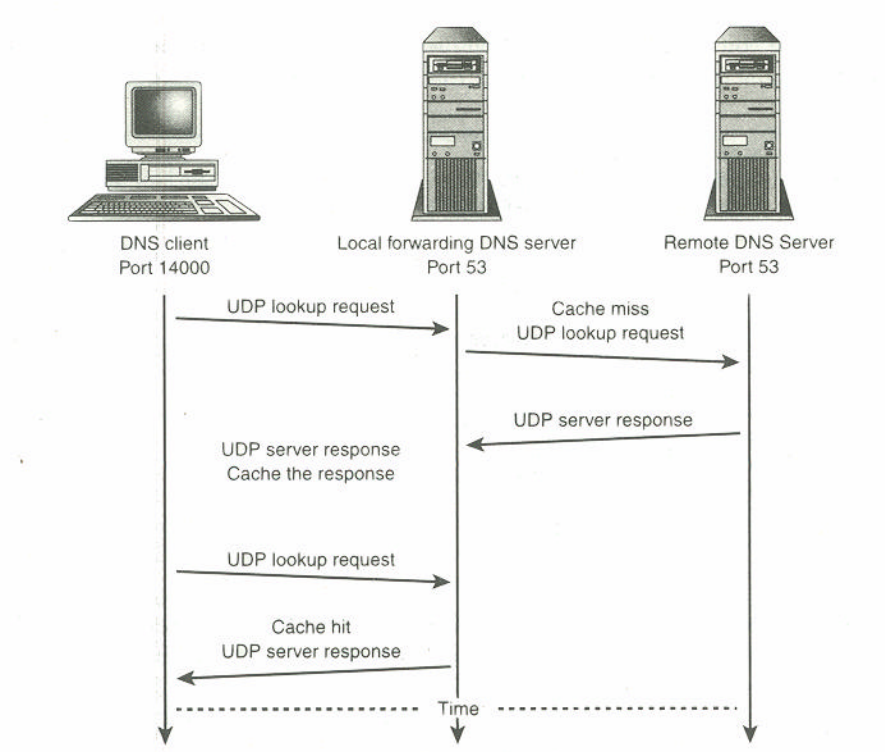
```
$IPTABLES -A FORWARD -p TCP \
-i $INTERNAL_INTERFACE \
–d $NAMESERVER –dport 53 \
–j ACCEPT

$IPTABLES –A FORWARD –p TCP –m state \
--state ESTABLISHED \
-j ACCEPT
```

Allowing Your DNS Lookups as a Peer-to-Peer Server

Peer-to-peer transactions are exchanges between two servers. In the case of DNS, when your local name server doesn't have the information a client requested stored locally, it contacts a remote server and forwards the request to it.

Configuring a local forwarding name server can be a big performance gain. As shown in the figure below, when *named* is configured as a caching and forwarding name server, it functions both as local sever and as a client to a remote DNS server. The difference between a direct client-to-server exchange and a forwarded local server-to-remote server exchange (peer-to-peer) are:

1) Instead of initiating an exchange from an unprivileged port, *named* initiates the exchange from its own DNS port 53.
2) Peer-to-peer server lookups of this type are always done over UDP.



A DNS forwarding server and peer-to-peer lookup.

Local client requests are sent to the local DNS server. The first time, *named* won't have the lookup information, so it forwards the request to a remote name server. *named* caches the returned information and passes it on to the client. The next time the same information is requested, *named* finds it in its local cache and doesn't make a remote request. The rules allowing this type of communication are:

```
$IPTABLES -A FORWARD -p UDP \
-i $INTERNAL_INTERFACE \
-s $LOCAL_NAMESERVER -sport 53 \
–d $REMOTE_NAMESERVER –dport 53 \
–j ACCEPT

$IPTABLES –A FORWARD –p UDP –m state \
--state ESTABLISHED \
-j ACCEPT
```

Allowing Remote DNS Lookups to Your Server

The average home-based site has little reason to provide DNS service to remote machines. Unless your site is an ISP, your clients will be machines local to your LAN, your subnet, your router, your network address space, or whatever terms are in use for the size of a network and the organization you're looking at.

Assuming you are a home or small business offering DNS to the outside world, you would limit the clients to a select group. You would not allow connections from just anywhere.

```
# client-to-server DNS transaction
$IPTABLES -A FORWARD -p UDP \
-i $EXTERNAL_INTERFACE \
-s $MY_DNS_CLIENTS \
-d $LOCAL_NAMESERVER -dport 53 \
-j ACCEPT

$IPTABLES –A FORWARD –p UDP –m state \
--state ESTABLISHED \
-j ACCEPT


# peer-to-peer server DNS transaction
$IPTABLES -A FORWARD -p UDP \
-i $EXTERNAL_INTERFACE \
-s $MY_DNS_CLIENTS -sport 53 \
-d $LOCAL_NAMESERVER -dport 53 \
-j ACCEPT

$IPTABLES –A FORWARD –p UDP –m state \
--state ESTABLISHED \
-j ACCEPT
```

2) Allowing the AUTH User Identification Service (TCP Port 113)

The AUTH, or IDENTD, user identification service is most often used when sending mail or posting Usenet article. Some FTP sites are also configured to require a resolvable AUTH lookup. For logging purposes, the server initiates an AUTH request back to your machine to get the account name of the user who initiated the mail or news connection. The table below lists the complete client/server connection protocol for the AUTH service.

***identd* Protocol**

Description	Protocol	Remote Address	Remote Port	In/Out	Local Address	Local Port	TCP flag
Local client query	TCP	ANYWHERE	113	Out	IPADDR	1024:65535	Any
Remote Server response	TCP	ANYWHERE	113	In	IPADDR	1024:65535	Ack
Local client query	TCP	ANYWHERE	1024:65535	In	IPADDR	113	Any
Local server response	TCP	ANYWHERE	1024:65535	Out	IPADDR	113	ACK

Allowing Your Outgoing AUTH Requests as a Client

Your machine will act as an AUTH client is you ran a mail or FTP server. There is no reason not to allow your system to be an AUTH client.

```
$IPTABLES -A FORWARD -p TCP \
-i $INTERNAL_INTERFACE \
-dport 113 \
-j ACCEPT

$IPTABLES –A FORWARD –p TCP –m state \
--state ESTABLISHED \
-j ACCEPT
```

Filtering Incoming AUTH Requests to Your Server

Offering AUTH services is a matter of debate. There appear to be no overwhelming arguments to make the case for either side, other than that a growing number of FTP sites require it, and AUTH provides user account information.

```
$IPTABLES -A FORWARD -p TCP \
-i $EXTERNAL_INTERFACE \
-dport 113 \
-j ACCEPT

$IPTABLES –A FORWARD –p TCP –m state \
--state ESTABLISHED \
-j ACCEPT
```

If you decide not to offer the service, then the result would be a long wait each time you tried to send mail or post a Usenet article. Your mail client won't be notified that the mail or article was received for delivery until the *identd* request timed out. Indeed, you need to reject the connection request to avoid waiting for TCP connection timeout.

```
$IPTABLES -A FORWARD -p TCP \
-i $EXTERNAL_INTERFACE \
-dport 113 \
-j REJECT
```

Enabling Common TCP Services

Possibly no one will want to enable all the services listed in this section, but everyone will want to enable some subset of them. These are the services most often used over the Internet today. This section provides rules for the following:

- Email
- Usenet

- telnet
- ssh
- ftp
- Web services
- finger
- whois
- gopher Information Service
- Wide Area Information Service (WAIS)

1) Email (TCP SMTP Port 25, POP Port 110, IMAP Port 143)

Email is the service almost everyone wants. How mail is set up depends upon your ISP, your connection type, and your own choices. Email is sent across the network using the SMTP protocol assigned to TCP service port 25. Email is commonly received locally through one of three different protocols - SMTP, POP, or IMAP - depending on the services your ISP provides and on your local configuration.

SMTP is the general mail protocol. Mail is delivered to the destination host machine. The endpoint mail server determines whether the mail is deliverable (addressed to a valid user account on the machine) and delivers it to the user's local mailbox.

POP and IMAP are mail retrieval services. POP runs on TCP port 110. IMAP runs on TCP port 143. ISPs commonly make incoming mail available to their customers using one of these two services. Both services are authenticated. They are associated with the ISP customer's user account and password. As far as mail retrieval is concerned, the difference between SMTP and POP or IMAP is that SMTP receives incoming mail and queues it in the user's local mailbox. POP and IMAP retrieve mail into the user's local mail program from the use's ISP, where the mail had been queued remotely in the user's SMTP mailbox at the ISP. The table below lists the complete client/server connection protocols for SMTP, POP and IMAP.

SMTP, POP and IMAP Mail Protocols

Description	Protocol	Remote Address	Remote Port	In/Out	Local Address	Local Port	TCP flag
Send outgoing mail	TCP	ANYWHERE	25	Out	IPADDR	1024:65535	Any
Remote server response	TCP	ANYWHERE	25	In	IPADDR	1024:65535	Ack
Receive incoming mail	TCP	ANYWHERE	1024:65535	In	IPADDR	25	Any
Local server response	TCP	ANYWHERE	1024:65535	Out	IPADDR	25	Ack
Local client query	TCP	POP SERVER	110	Out	IPADDR	1024:65535	Any
Remote server response	TCP	POP SERVER	110	In	IPADDR	1024:65535	Ack
Remote client query	TCP	POP Client	1024:65535	In	IPADDR	110	Any
Local server response	TCP	POP Client	1024:65535	Out	IPADDR	110	Ack
Local client query	TCP	IMAP SERVER	143	Out	IPADDR	1024:65535	Any
Remote server response	TCP	IMAP SERVER	143	In	IPADDR	1024:65535	Ack
Remote client query	TCP	IMAP Client	1024:65535	In	IPADDR	143	Any
Local server response	TCP	IMAP Client	1024:65535	Out	IPADDR	143	Ack

Sending Mail Over SMTP (TCP Port 25)

Mail is sent over SMTP. But whose SMTP server do you use to collect your mail and send it onward? ISPs offer SMTP mail service to their customers. The ISP's mail server acts as the mail gateway. It knows how to collect your mail, find the recipient host, and relay the mail.

Relaying Outgoing Mail Through an External (ISP) Gateway SMTP Server

When you relay outgoing mail through an external gateway SMTP server, your client mail program sends all outgoing mail to your ISP's mail server. Your ISP acts as your mail gateway to the rest of the world. Your system doesn't need to know how to locate your mail destinations or the routes to them. The ISP mail gateway serves as your relay.

The following rules allow you to relay mail though your ISP's SMTP gateway:

```
$IPTABLES -A FORWARD -p TCP \
-i $INTERNAL_INTERFACE \
-s $MY_INTERNAL_IPADDRS \
-o $EXTERNAL_INTERFACE \
-d $SMTP_GATEWAY -dport 25 \
-j ACCEPT
```

```
$IPTABLES –A FORWARD –p TCP –m state \
--state ESTABLISHED \
-j ACCEPT
```

Sending Mail to Any External Mail Server

Alternatively, you can bypass your ISP's mail server and host your own. Your local server is responsible for collecting your outgoing mail, doing the DNS lookup on the destination hostname, and relaying the mail to its destination. Your client mail program points to your local SMTP server rather than to the ISP's server.

The following rules allow you to send mail mail directly to the remote destinations:

```
$IPTABLES -A FORWARD -p TCP \
-i $INTERNAL_INTERFACE \
-s $MY_INTERNAL_IPADDRS \
-o $EXTERNAL_INTERFACE \
-dport 25 \
-j ACCEPT

$IPTABLES –A FORWARD –p TCP –m state \
--state ESTABLISHED \
-j ACCEPT
```

Receiving Mail

How you receive mail depends on your situation. If you run your own local mail server, you can collect incoming mail directly on your machine. If you retrieve your mail from your ISP account, you may or may not retrieve mail as a POP or IMAP client, depending on how you've configured your ISP email account, and depending on the mail delivery services the ISP offers.

Receiving Mail as a Local SMTP Server (TCP Port 25)

If you want to receive mail sent directly to your local machines from anywhere in the world, you need to run a SMTP server and use these server rules:

```
$IPTABLES -A FORWARD -p TCP \
-i $EXTERNAL_INTERFACE \
-o $INTERNAL_INTERFACE \
-d $MY_INTERNAL_IPADDRS \
-dport 25 \
-j ACCEPT

$IPTABLES –A FORWARD –p TCP –m state \
--state ESTABLISHED \
-j ACCEPT
```

Alternatively, if you'd rather keep your local email account relatively private and use your work or ISP email account as your public address, you could configure your work and ISP mail accounts to forward mail to your local server. In this case, you could replace the previous single rule pair, accepting connections from anywhere, with separate, specific rules for each mail forwarder.

Retrieving Mail as a POP Client (TCP Port 110)

Connecting to a POP server is a very common means of retrieving mail from a remote ISP or work account. If your ISP uses POP server for customer mail retrieval, you need to allow outgoing client-to-server connections.

The server's address will be a specific hostname or address, rather than the global ANYWHERE specifier. POP accounts are user accounts associated with a specific user and password:

```
$IPTABLES -A FORWARD -p TCP \
-i $INTERNAL_INTERFACE \
-o $EXTERNAL_INTERFACE \
-d $POP_SERVER -dport 110 \
-j ACCEPT

$IPTABLES –A FORWARD –p TCP –m state \
--state ESTABLISHED \
-j ACCEPT
```

Receiving Mail as an IMAP Client (TCP Port 143)

Connecting to an IMAP server is another common means of retrieving mail from a remote ISP or work account. If your ISP uses an IMAP server for customer mail retrieval, you need to allow outgoing client-to-server connections.

The server's address will be a specific hostname or address, rather than the global ANYWHERE specifier. IMAP accounts are user accounts associated with a specific user and password:

```
$IPTABLES -A FORWARD -p TCP \
-i $INTERNAL_INTERFACE \
-o $EXTERNAL_INTERFACE \
-d $IMAP_SERVER -dport 143 \
-j ACCEPT

$IPTABLES –A FORWARD –p TCP –m state \
--state ESTABLISHED \
-j ACCEPT
```

Hosting a Mail Server for Remote Clients

Hosting public POP or IMAP services is unusual for a small system. It's important to limit the clients your system will accept connections from, both on the packet-filtering level and on the server configuration level. You should also consider using an encrypted authentication method, or allow mail retrieval only over an SSH connection.

Hosting a POP Server for Remote Clients

POP servers are one of the three most common and successful points of entry for hacking exploits.

If you use a local system as a central mail server and run a local server to provide mail access to local machines on a LAN, you don't need the server rules in this example. Incoming connections from the Internet should be denied. If you do need to host POP service for a limited number of remote individuals, the next two rules allow incoming connections to your POP server. Connections are limited to your specific clients' IP address:

```
$IPTABLES -A FORWARD -p TCP \
-i $EXTERNAL_INTERFACE \
-s <my.pop.clients> \
-d $POP_SERVER -dport 110 \
-j ACCEPT

$IPTABLES –A FORWARD –p TCP –m state \
--state ESTABLISHED \
-j ACCEPT
```

Hosting an IMAP Server for Remote Clients

IMAP servers are one of the three most common and successful points of entry for hacking exploits.

If you use a local system as a central mail server and run a local server to provide mail access to local machines on a LAN, you don't need a server rule. Incoming connections from the Internet should be denied. If you do need to host IMAP service for a limited number of remote individuals, the next two rules allow incoming connections to your IMAP server. Connections are limited to your specific clients' IP addresses:

```
$IPTABLES -A FORWARD -p TCP \
-i $EXTERNAL_INTERFACE \
-s <my.imap.clients> \
-d $IMAP_SERVER -dport 143 \
-j ACCEPT

$IPTABLES –A FORWARD –p TCP –m state \
--state ESTABLISHED \
-j ACCEPT
```

2) Accessing Usenet News Services (TCP Port 119)

Usenet news is accessed over NNTP running on top of TCP through service port 119. Your local news client handles reading news and posting articles. Few systems require the server rules. The table below lists the complete client/server connection protocols for the NNTP Usenet news service.

NNTP Protocol

Description	Protocol	Remote Address	Remote Port	In/Out	Local Address	Local Port	TCP flag
Local client query	TCP	NEWS SERVER	119	Out	IPADDR	1024:65535	Any
Remote server response	TCP	NEWS SERVER	119	In	IPADDR	1024:65535	Ack
Remote client query	TCP	NNTP Clients	1024:65535	In	IPADDR	119	Any
Local server response	TCP	NNTP Clients	1024:65535	Out	IPADDR	119	Ack
Local server query	TCP	News feed	119	Out	IPADDR	1024:65535	Any
Remote server response	TCP	News feed	119	In	IPADDR	1024:65535	Ack

Reading and Posting News as a Usenet Client

The client rules allow connections to your ISP's news server. Both reading news and posting articles are handled by these rules.

```
$IPTABLES -A FORWARD -p TCP \
-o $EXTERNAL_INTERFACE \
-d $NEWS_SERVER -dport 119 \
-j ACCEPT

$IPTABLES –A FORWARD –p TCP –m state \
--state ESTABLISHED \
-j ACCEPT
```

Hosting a Usenet News Server for Remote Clients

A small site is very unlikely to host a news server for the outside world. Even hosting a local news server is unlikely. For the rare exception, the server rules should be configured to allow incoming connections only from a select set of clients:

```
$IPTABLES -A FORWARD -p TCP \
-i $EXTERNAL_INTERFACE \
-s <my.news.clients> \
-d $LOCAL_NEWS_SERVER -dport 119 \
-j ACCEPT
```

```
$IPTABLES –A FORWARD –p TCP –m state \
--state ESTABLISHED \
-j ACCEPT
```

Allowing Peer News Feeds for a Local Usenet Server

A small, home-based site is unlikely to have a peer-to-peer news feed server relationship with an ISP. Although news servers used to be fairly accessible to the general Internet, few open news servers are available anymore due to SPAM and server load issues.

If your site is large enough or rich enough to host a general Usenet server, you have to get your news feed from somewhere. The next two rules allow your local news server to receive its news feed from a remote server. The local server contacts the remote server as a client. The only difference between the peer-to-peer news feed rules and the regular client rules is the name or address of the remote host:

```
$IPTABLES -A FORWARD -p TCP \
-s $LOCAL_NEWS_SERVER  \
-o $EXTERNAL_INTERFACE \
-d <my.news.feed> -dport 119 \
-j ACCEPT

$IPTABLES –A FORWARD –p TCP –m state \
--state ESTABLISHED \
-j ACCEPT
```

3) Telnet (TCP Port 119)

telnet has been the de facto standard means of remote login over the Internet for many years. As the nature of Internet community has changed, *telnet* has come to be viewed more and more as an insecure service, because it communicates in ASCII clear text. Nevertheless, *telnet* may be the only tool available to you for remote connections, depending on the connection options available at the other end. If you have the option, you should always use an encrypted service, such as *ssh*, rather than *telnet*.

The client and server rules here allow access to and from anywhere. If you use *telnet*, you can probably limit the external addresses to a select subset at the packet-filtering level. The table below lists the complete client/server connection protocols for the *telnet* service.

Telnet Protocol

Description	Protocol	Remote Address	Remote Port	In/Out	Local Address	Local Port	TCP flag
Local client request	TCP	ANYWHERE	23	Out	IPADDR	1024:65535	Any
Remote server response	TCP	ANYWHERE	23	In	IPADDR	1024:65535	Ack
Remote client request	TCP	*telnet* clients	1024:65535	In	IPADDR	23	Any
Local server response	TCP	*telnet* clients	1024:65535	Out	IPADDR	23	Ack

Allowing Outgoing Client Access to Remote Sites

If you need to use *telnet* to access your accounts on remote systems, the next two rules allow outgoing connections to remote sites. If your site has multiple users, you might limit outgoing connections to the specific sites your users have accounts on, rather than allowing outgoing connections to ANYWHERE:

```
$IPTABLES -A FORWARD -p TCP \
-o $EXTERNAL_INTERFACE \
-dport 119\
-j ACCEPT

$IPTABLES –A FORWARD –p TCP –m state \
--state ESTABLISHED \
-j ACCEPT
```

Allowing Incoming Access to Your Local Server

Even if you need client access to remote servers, you may not need to allow incoming connections to your *telnet* server. If you do, the next two rules allow incoming connections to your server:

```
$IPTABLES -A FORWARD -p TCP \
-i $EXTERNAL_INTERFACE \
-d $TELNET_SERVER -dport 119\
-j ACCEPT

$IPTABLES –A FORWARD –p TCP –m state \
--state ESTABLISHED \
-j ACCEPT
```

4) *ssh* (TCP Port 22)

SSH is considered far preferable to using *telnet* for remote login access, because both ends of the connection use authentication keys for both hosts and users, and data is encrypted. Additionally, SSH is more than a remote login service. It can automatically direct X Window connections between remote sites, and FTP and other TCP-based connections can be directed over the more secure SSH connection.

Provided that the other end of the connection allows SSH connections, it's possible to route all TCP connections through the firewall using SSH. As such, SSH is something of a poor man's virtual private network (VPN).

The ports used by SSH are highly configurable. The rules, given later, apply to the default SSH port usage. By default, connections are initiated between a client's unprivileged port and the server's assigned service port 22. The server forks off a copy of itself for the connection, and the client end of the connection is then reassigned to a privileged port in the descending range from 1023 to 513 in order to support *.rhosts* and *hosts.equiv* authentication. The first available port is used. The SSH client will optionally use the unprivileged ports exclusively. The SSH server will accept connections from either the privileged or unprivileged ports.

The client and server rules here allow access from anywhere. In practice, you would limit the external addresses to a select subset, particularly because both ends of the connection must be configured to recognize each individual user account for authentication. The table below lists the complete client/server connection protocol for the SSH service.

SSH Protocol

Description	Protocol	Remote Address	Remote Port	In/Out	Local Address	Local Port	TCP flag
Local client request	TCP	ANYWHERE	22	Out	IPADDR	1024:65535	Any
Remote server response	TCP	ANYWHERE	22	In	IPADDR	1024:65535	Ack
Local client request	TCP	ANYWHERE	22	Out	IPADDR	513:1023	Any
Remote server response	TCP	ANYWHERE	22	In	IPADDR	513:1023	Ack
Remote client request	TCP	SSH clients	1024:65535	In	IPADDR	22	Any
Local server response	TCP	SSH clients	1024:65535	Out	IPADDR	22	Ack
Remote client request	TCP	SSH clients	513:1023	In	IPADDR	22	Any
Local server response	TCP	SSH clients	513:1023	Out	IPADDR	22	Ack

When selecting a privileged server port for the ongoing connection, the first free port between 1023 and 513 is used. The range of ports you allow equates to the number of simultaneous incoming SSH connections you allow.

Allowing Client Access to Remote SSH Servers

```
# SSH_PORTS are the range of ports you allow from
privileged ports

$IPTABLES -A FORWARD -p TCP \
```

```
-o $EXTERNAL_INTERFACE \
-dport 22 \
-j ACCEPT

$IPTABLES –A FORWARD –p TCP –m state \
--state ESTABLISHED \
-j ACCEPT

$IPTABLES -A FORWARD -p TCP \
-sport $SSH_PORTS \
-o $EXTERNAL_INTERFACE \
-dport 22\
-j ACCEPT

$IPTABLES –A FORWARD –p TCP –m state \
--state ESTABLISHED \
-j ACCEPT
```

Allowing Remote Client Access to your Local SSH Server

```
# SSH_PORTS are the range of ports you allow from
privileged ports

$IPTABLES -A FORWARD -p TCP \
-i $EXTERNAL_INTERFACE \
-d $SSH_SERVER -dport 22 \
-j ACCEPT

$IPTABLES –A FORWARD –p TCP –m state \
--state ESTABLISHED \
-j ACCEPT

$IPTABLES -A FORWARD -p TCP \
-i $EXTERNAL_INTERFACE \
-sport $SSH_PORTS \
-d $SSH_SERVER -dport 22 \
-j ACCEPT

$IPTABLES –A FORWARD –p TCP –m state \
--state ESTABLISHED \
-j ACCEPT
```

5) *ftp* (TCP Port 20, 21)

FTP remains one of the most common means of transferring files between two networked machines. Web-based interfaces to FTP have become common, as well.

FTP uses two privileged ports, one for sending commands and one for sending data. Port 21 is used to establish the initial connection to the server and pass user commands. Port 20 is used to establish a data channel over which files and directory listings are sent as data.

FTP has two modes for exchanging data between a client and server, normal data channel port mode and passive data channel mode. Normal port mode is the original, default mechanism when using the ftp client program and connecting to a remote FTP site. Passive mode is a newer mechanism, and is the default when connecting through a Web browser. Occasionally, you might encounter an FTP site that supports only one mode or the other. The table below lists the complete client/server connection protocol for the FTP service.

FTP Protocol

Description	Protocol	Remote Address	Remote Port	In/Out	Local Address	Local Port	TCP flag
Local client query	TCP	ANYWHERE	21	Out	IPADDR	1024:65535	Any
Remote server response	TCP	ANYWHERE	21	In	IPADDR	1024:65535	Ack
Remote server port data channel request	TCP	ANYWHERE	20	In	IPADDR	1024:65535	Any
Local client port data channel response	TCP	ANYWHERE	20	Out	IPADDR	1024:65535	Ack
Local client passive data channel request	TCP	ANYWHERE	1024:65535	Out	IPADDR	1024:65535	Any
Remote server passive data channel response	TCP	ANYWHERE	1024:65535	In	IPADDR	1024:65535	Ack
Remote client request	TCP	ANYWHERE	1024:65535	In	IPADDR	21	Any
Local server response	TCP	ANYWHERE	1024:65535	Out	IPADDR	21	Ack
Remote client port data channel response	TCP	ANYWHERE	1024:65535	In	IPADDR	20	Ack
Local server port data channel request	TCP	ANYWHERE	1024:65535	Out	IPADDR	20	Any
Remote client passive data channel request	TCP	ANYWHERE	1024:65535	In	IPADDR	1024:65535	Any
Local server passive data channel response	TCP	ANYWHERE	1024:65535	Out	IPADDR	1024:65535	Ack

The unusual callback behavior, where the remote server establishes the secondary connection with your client, is part of what makes FTP difficult to track, and thereby difficult to secure, at the packet-filtering level. Therefore the connection-tracking module (ip_conntrack) needs a helper module (ip_conntrack_ftp) to track FTP connections. This module should be loaded in order to correctly track the FTP connections.

Allowing Outgoing Client Access to Remote FTP Servers

It's almost given that most sites will want FTP client access to remote file repositories. Most people will want to enable outgoing client connections to a remote server.

Outgoing FTP Requests

The next two rules allow an outgoing connection to a remote FTP server. The state RELATED is used here in order to allow those packets that have been related with this connection and identified by the connection-tracking module.

```
$IPTABLES -A FORWARD -p TCP \
-o $EXTERNAL_INTERFACE \
-dport 21 \
-j ACCEPT

$IPTABLES –A FORWARD –p TCP –m state \
--state ESTABLISHED,RELATED \
-j ACCEPT
```

Normal Port Mode FTP Data Channels

For IPTABLES to correctly track the FTP data channel connection you need to load the module *ip_conntrack_ftp.o*

```
insmod ip_conntrack_ftp.o
```

Passive Mode FTP Data Channels

Passive mode is considered more secure than normal port mode because the FTP client initiates both the control and data connections, even tough the connection is made between two unprivileged ports. But, as we don't know in advance which ports are to be used, therefore we need to leave this work to the FTP helper module for connection tracking (*ip_conntrack_ftp*).

Allowing Incoming Access to Your Local FTP Server

Whether to offer FTP services to the world is a difficult decision. Although FTP sites abound on the Internet, FTP server configuration requires great care. Numerous FTP security exploits are possible.

If your goal is to offer general read-only access to some set of files on your machine, you might consider making these files available through a Web server. If your goal is to allow file uploads to your machine from the outside, FTP server access should be severely limited on the firewall level, on the *tcp_wrapper* level, and on the FTP configuration level.

In any case, if you decide to offer FTP services, and if you decide to allow incoming file transfers, write access should not be allowed via Anonymous FTP.

Remote write access to your file systems should be allowed only from specific, authenticated FTP user accounts, from specific remote sites, and to carefully control and limit FTP areas reserved in your file system.

Incoming FTP Requests

The next two rules allow incoming connections to your FTP server.

```
$IPTABLES -A FORWARD -p TCP \
-i $EXTERNAL_INTERFACE \
-d $LOCAL_FTP_SERVER -dport 21 \
-j ACCEPT

$IPTABLES –A FORWARD –p TCP –m state \
--state ESTABLISHED,RELATED \
-j ACCEPT
```

Normal Port Mode FTP Data Channel Responses

For IPTABLES to correctly track the FTP data channel connection you need to load the module *ip_conntrack_ftp.o*

```
insmod ip_conntrack_ftp.o
```

Passive Mode FTP Data Channel Responses

For IPTABLES to correctly track the FTP data channel connection you need to load the module *ip_conntrack_ftp.o*

```
insmod ip_conntrack_ftp.o
```

6) Web Services

Web services are based on Hypertext Transfer Protocol (HTTP). Client and server connections use the standard TCP conventions. Several higher-level, special-purpose communication protocols are available, in addition to the standard general HTTP access, including secure access over SSL, and access via an ISP-provided Web server proxy. These different access protocols use different service ports.

Standard HTTP Access (TCP Port 80)

In normal use, Web services are available over HTTP service port 80. The table below lists the complete client/server connection protocol for the HTTP Web service.

HTTP Protocol

Description	Protocol	Remote Address	Remote Port	In/Out	Local Address	Local Port	TCP flag
Local client request	TCP	ANYWHERE	80	Out	IPADDR	1024:65535	Any
Remote server response	TCP	ANYWHERE	80	In	IPADDR	1024:65535	Ack
Remote client request	TCP	ANYWHERE	1024:65535	In	IPADDR	80	Any
Local server response	TCP	ANYWHERE	1024:65535	Out	IPADDR	80	Ack

Accessing Remote Web Sites as a Client

It's almost inconceivable in today's world that a home-based site would not want to access the World Wide Web from a Web browser. The next two rules allow access to remote Web servers.

```
$IPTABLES -A FORWARD -p TCP \
-o $EXTERNAL_INTERFACE \
-dport 80 \
-j ACCEPT

$IPTABLES –A FORWARD –p TCP –m state \
--state ESTABLISHED \
-j ACCEPT
```

Allowing Remote Access to a Local Web Server

If you decide to run a Web server of your own and host a Web site for the Internet, the general server rules allow all typical incoming access to your site.

```
$IPTABLES -A FORWARD -p TCP \
-i $EXTERNAL_INTERFACE \
-d $LOCAL_WEB_SERVER -dport 80 \
-j ACCEPT

$IPTABLES –A FORWARD –p TCP –m state \
--state ESTABLISHED \
-j ACCEPT
```

Secure Web Access (SSL) (TCP Port 443)

Secure Socket Layer (SSL) is used for secure, encrypted Web access. The SSL protocol uses TCP port 443. You will most often encounter this if you go to a commercial Web site to purchase something, use online banking services, or enter a protected Web area where you'll be prompted for personal information. The table below lists the complete client/server connection protocol for the SSL service.

SSL Protocol

Description	Protocol	Remote Address	Remote Port	In/Out	Local Address	Local Port	TCP flag
Local client request	TCP	ANYWHERE	443	Out	IPADDR	1024:65535	Any
Remote server response	TCP	ANYWHERE	443	In	IPADDR	1024:65535	Ack
Remote client request	TCP	ANYWHERE	1024:65535	In	IPADDR	443	Any
Local server response	TCP	ANYWHERE	1024:65535	Out	IPADDR	443	Ack

Accessing Remote Web Sites Over SSL as a Client

Most people will want client access to secure Web sites at some point or another.

```
$IPTABLES -A FORWARD -p TCP \
-o $EXTERNAL_INTERFACE \
-dport 443 \
-j ACCEPT

$IPTABLES –A FORWARD –p TCP –m state \
--state ESTABLISHED \
-j ACCEPT
```

Allowing Remote Access to a Local SSL Web Server

If you conduct some form of e-commerce, you'll mostly likely want to allow incoming connections to SSL-protected areas of your Web site.

```
$IPTABLES -A FORWARD -p TCP \
-i $EXTERNAL_INTERFACE \
-d $LOCAL_WEB_SERVER -dport 443 \
-j ACCEPT

$IPTABLES –A FORWARD –p TCP –m state \
--state ESTABLISHED \
-j ACCEPT
```

Web Proxy Access (TCP Ports 8008, 8080)

Publicly accessible Web server proxies are the most common at ISPs. As a customer, you configure your browser to use a remote proxy service. Web proxies are often accessed through one of two unprivileged ports assigned for this purpose, port 8008 or 8080, as defined by the ISP. In return, you get faster Web

page access when the pages are already cached locally at your ISP's server, and the anonymity of proxied access to remote sites. Your connections are not direct, but instead are initiated on your behalf by your ISP's proxy. The table below lists the local client to remote server connection protocol for the Web proxy service.

Web Proxy Protocol

Description	Protocol	Remote Address	Remote Port	In/Out	Local Address	Local Port	TCP flag
Local client request	TCP	WEB PROXY SERVER	WEB PROXY PORT	Out	IPADDR	1024:65535	Any
Remote server response	TCP	WEB PROXY SERVER	WEB PROXY PORT	In	IPADDR	1024:65535	Ack

Allowing Client Access to Remote Web Proxy Server

```
$IPTABLES -A FORWARD -p TCP \
-o $EXTERNAL_INTERFACE \
-d $WEB_PROXY_SERVER -dport $WEB_PROXY_PORT \
-j ACCEPT

$IPTABLES –A FORWARD –p TCP –m state \
--state ESTABLISHED \
-j ACCEPT
```

Allowing Remote Access to Local Web Proxy Server

If you decide to run a Web proxy server and provide the service to some of your clients, then the following rules will allow the remote clients to use this service.

```
$IPTABLES -A FORWARD -p TCP \
-i $EXTERNAL_INTERFACE \
-s <my.webproxy.clients>
-d $LOCAL_WEB_PROXY_SERVER -dport $WEB_PROXY_PORT \
-j ACCEPT

$IPTABLES –A FORWARD –p TCP –m state \
--state ESTABLISHED \
-j ACCEPT
```

7) Finger (TCP Port 79)

From a connection point of view, the *finger* service is harmless. Due to the changing nature of privacy issues in relation to a growing and changing Internet community, offering *finger* service is generally discouraged today. *finger* provides user account information, including such things as user login name, real name,

currently active logins, pending mail, and mail forwarding locations. Often *finger* provides user-furnished (in a *.plan* file) personal information as well, including phone numbers, home addresses, tasks and plans, vacation status, and so forth. The table below lists the complete client/server connection protocol for the finger service.

***finger* Protocol**

Description	Protocol	Remote Address	Remote Port	In/Out	Local Address	Local Port	TCP flag
Local client request	TCP	ANYWHERE	79	Out	IPADDR	1024:65535	Any
Remote server response	TCP	ANYWHERE	79	In	IPADDR	1024:65535	Ack
Remote client request	TCP	*finger* clients	1024:65535	In	IPADDR	79	Any
Local server response	TCP	*finger* clients	1024:65535	Out	IPADDR	79	Ack

Accessing Remote *finger* Servers as a Client

There's no harm is enabling outgoing access to remote *finger* servers, and these are the rules to do so:

```
$IPTABLES -A FORWARD -p TCP \
-o $EXTERNAL_INTERFACE \
-dport 79 \
-j ACCEPT

$IPTABLES –A FORWARD –p TCP –m state \
--state ESTABLISHED \
-j ACCEPT
```

Allowing Remote Client Access to a Local *finger* Server

If you choose to allow outside access to your *finger* service, it's recommended that you limit access to specific client sites. *finger* access can be limited both at the firewall level ant at the *tcp_wrapper* level.

The next two rules allow incoming connections to your *finger* server, but only selected hosts are allowed to initiate connection.

```
$IPTABLES -A FORWARD -p TCP \
-i $EXTERNAL_INTERFACE \
-s <my.finger.clients> \
-d $LOCAL_FINGER_SERVER -dport 79 \
-j ACCEPT

$IPTABLES –A FORWARD –p TCP –m state \
--state ESTABLISHED \
```

```
-j ACCEPT
```

8) *whois* (TCP Port 43)

The *whois* program accesses the InterNIC Registration Services database. it allows IP address and host and domain name lookups in human-readable form. The table below lists the local client to remote server connection protocol for the *whois* service.

***whois* Protocol**

Description	Protocol	Remote Address	Remote Port	In/Out	Local Address	Local Port	TCP flag
Local client request	TCP	ANYWHERE	43	Out	IPADDR	1024:65535	Any
Remote server response	TCP	ANYWHERE	43	In	IPADDR	1024:65535	Ack

Allowing Client Access to a Remote *whois* Server

The next two rules allow you to query an official remote server:

```
$IPTABLES -A FORWARD -p TCP \
-o $EXTERNAL_INTERFACE \
-dport 43 \
-j ACCEPT

$IPTABLES –A FORWARD –p TCP –m state \
--state ESTABLISHED \
-j ACCEPT
```

9) *gopher* (TCP Port 70)

The *gopher* information service is still available for low-overhead ASCII terminals, but Web-based search engines and the hypertext links have largely replaced its use. It is unlikely that you would offer local *gopher* service instead of a Web site. The table below lists the local client to remote server connection protocol for the *gopher* service.

***gopher* Protocol**

Description	Protocol	Remote Address	Remote Port	In/Out	Local Address	Local Port	TCP flag
Local client request	TCP	ANYWHERE	70	Out	IPADDR	1024:65535	Any
Remote server response	TCP	ANYWHERE	70	In	IPADDR	1024:65535	Ack

Allowing Client Access to Remote Gopher Server

The next two rules allow you to connect to a remote GOPHER server.

```
$IPTABLES -A FORWARD -p TCP \
-o $EXTERNAL_INTERFACE \
-dport 70 \
-j ACCEPT

$IPTABLES –A FORWARD –p TCP –m state \
--state ESTABLISHED \
-j ACCEPT
```

10) WAIS (TCP Port 210)

Wide Area Information Servers (WAIS) are now known as search engines. Web-browsers typically provide a graphical front-end to WAIS. Netscape contains the WAIS client code necessary to connect to WAIS. The table below lists the local client to remote server connection protocol for the WAIS service.

WAIS Protocol

Description	Protocol	Remote Address	Remote Port	In/Out	Local Address	Local Port	TCP flag
Local client request	TCP	ANYWHERE	210	Out	IPADDR	1024:65535	Any
Remote server response	TCP	ANYWHERE	210	In	IPADDR	1024:65535	Ack

Allowing Client Access to Remote WAIS Server

The next two rules allow you to connect to a remote WAIS server.

```
$IPTABLES -A FORWARD -p TCP \
-o $EXTERNAL_INTERFACE \
-dport 210 \
-j ACCEPT

$IPTABLES –A FORWARD –p TCP –m state \
--state ESTABLISHED \
-j ACCEPT
```

Enabling Common UDP Services

The stateless UDP protocol is inherently less secure than the connection-based TCP protocol. Because of this, many security-conscious sites completely disable, or else limit as much as possible, all the access to UDP services. Obviously, UDP-based DNS exchanges are necessary, but the remote name servers can be explicitly

specified in the firewall rules. As such, this section provides rules for the following services:

- *traceroute*
- Dynamic Host Configuration Protocol (DHCP)
- Network Time Protocol (NTP)

traceroute (UDP Port 33434)

traceroute is a UDP service that causes intermidiate systems to generate ICMP Time Exceeded messages to gather hop count information, and the target system to return a Destination Unreachable (port not found) message, indicating the endpoint of the route to the host. The table below lists the local client to remote server connection protocol for the *traceroute* service.

traceroute Protocol

Description	Protocol	Remote Address	Remote Port/ICMP Type	In/Out	Local Address	Local Port/ICMP Type
Outgoing *traceroute* probe	UDP	ANYWHERE	33434:33523	Out	IPADDR	32769:65535
Time Exceeded (intermediate hop)	UDP	ANYWHERE	11	In	IPADDR	-
Port not found (termination)	UDP	ANYWHERE	3	In	IPADDR	-
Incoming *traceroute* probe	UDP	ISP	32769:65535	In	IPADDR	33434:33523
Time Exceeded (intermediate hop)	UDP	ISP	-	Out	IPADDR	11
Port not found (termination)	UDP	ISP	-	Out	IPADDR	3

traceroute can be configured to use any port or port range. As such, it's difficult to block all incoming *traceroute* packets by listing specific ports. However, it often uses source ports in the range from 32769 to 65535 and destination ports in the range from 33434 to 33523.

Enabling Outgoing *traceroute* Requests

If you intent to use *traceroute* yourself, you must enable the UDP client ports. Note that you must allow incoming ICMP Time Exceeded and Destination Unreachable messages from anywhere for outgoing *traceroute* to work.

```
$TRACEROUTE_SRC_PORTS="32769:65535"
$TRACEROUTE_DEST_PORTS="33434:33523"

$IPTABLES -A FORWARD -p UDP \
-i $INTERNAL_INTERFACE
-sport $TRACEROUTE_SRC_PORTS
-o $EXTERNAL_INTERFACE \
-dport $TRACEROUTE_DEST_PORTS \
-j ACCEPT

$IPTABLES –A FORWARD –p UDP –m state \
--state ESTABLISHED \
-j ACCEPT

#allowing ICMP Time Exceeded and Destination
Unreachable messages from outside
$IPTABLES -A FORWARD -p ICMP \
-i $EXTERNAL_INTERFACE \
--icmp-type 11

$IPTABLES -A FORWARD -p ICMP \
-i $EXTERNAL_INTERFACE \
--icmp-type 3
```

Allowing Incoming *traceroute* Requests

Because *traceroute* is a less secure UDP service and can be used to attack other UDP services, the following example opens incoming *traceroute* from only your ISP. Note that you must allow outgoing ICMP Time Exceeded and Destination Unreachable messages to be targeted to your ISP for incoming *traceroute* to work.

```
$TRACEROUTE_SRC_PORTS="32769:65535"
$TRACEROUTE_DEST_PORTS="33434:33523"

$IPTABLES -A FORWARD -p UDP \
-i $EXTERNAL_INTERFACE
-s $MY_ISP -sport $TRACEROUTE_SRC_PORTS
-o $EXTERNAL_INTERFACE \
-dport $TRACEROUTE_DEST_PORTS \
-j ACCEPT

$IPTABLES –A FORWARD –p UDP –m state \
--state ESTABLISHED \
-j ACCEPT

#allowing ICMP Time Exceeded and Destination
Unreachable messages to go to your ISP
$IPTABLES -A FORWARD -p ICMP \
-o $EXTERNAL_INTERFACE \
-d $MY_ISP
```

```
--icmp-type 11

$IPTABLES -A FORWARD -p ICMP \
-o $EXTERNAL_INTERFACE \
-d $MY_ISP
--icmp-type 3
```

Accessing Your ISP's DHCP Server (UDP Ports 67, 68)

DHCP exchanges, if any, between your site and your ISP's server will necessarily be local client to remote server exchanges. DHCP clients receive temporary, dynamically allocated IP addresses from a central server that manages the ISP's customer IP address apace.

If you have a dynamically allocated IP address from your ISP, you need to run the DHCP client on your machine. It's not uncommon for bogus DHCP server messages to fly around your ISP's local subnet if someone runs the server by accident. For this reason, it's especially important to filter DHCP messages to limit traffic between your client and your specific ISP DHCP server as much as possible.

The table below lists the DHCP message type descriptions as quoted from RFC 2131, "Dynamic Host Configuration Protocol".

DHCP Message Types

DHCP Message	Description
DHCPDISCOVER	Client broadcast to locate available servers.
DHCPOFFER	Server to client in response to DHCPDISCOVER with offer of configuration parameters.
DHCPREQUEST	Client message to servers either (a) requesting offered parameters from one server and implicitly declining offers from all others; (b) confirming correctness of previously allocated address after e.g. system reboot; or (c) extending the lease on a particular network address.
DHCPACK	Server to client with configuration parameters, including commited network address.
DHCPNAK	Server to client indicating client's notion of network address is incorrect (e.g. client has moved to new subnet) or client's lease has expired.
DHCPDECLINE	Client to server indicating network address is already in use.
DHCPRELEASE	Client to server relinquishing network address and canceling remainig lease.
DHCPINFORM	Client to server, asking only for local configuration parameters; client already has externally configured address.

In essence, when the DHCP client initializes, it broadcasts a DHCPDISCOVER query to whether any DHCP servers are available. Any servers receiving the query may respond with a DHCPOFFER message indicating their willingness to

function as a server to this client, and include the configuration parameters they have to offer. The client broadcasts a DHCPREQUEST message to both accept one of the servers and inform any remaining servers that it has chosen to decline their offers. The chosen server responds with a DHCPACK message, indicating confirmation of the parameters it originally offered. Address assignment is complete at this point. Periodically, the client will send a DHCPREQUEST message requesting a renewal on the IP address lease. If the lease is renewed, the server responds with a DHCPACK message. Otherwise, the client falls back to the initialization process. The table below lists the local client to remote server exchange protocol for the DHCP. The DHCP protocol is far more complicated than this brief summary, but the summary describes the essentials of the typical client and server exchange.

DHCP Protocol

Description	Protocol	Remote Address	Remote Port	In/Out	Local Address	Local Port
DHCPDISCOVER; DHCPREQUEST	UDP	255.255.255.255	67	Out	0.0.0.0	68
DHCPOFFER	UDP	0.0.0.0	67	In	255.255.255.255	68
DHCPOFFER	UDP	DHCP SERVER	67	In	255.255.255.255	68
DHCPREQUEST; DHCPDECLINE	UDP	DHCP SERVER	67	Out	0.0.0.0	68
DHCPACK; DHCPNAK	UDP	DHCP SERVER	67	In	ISP/netmask	68
DHCPACK	UDP	DHCP SERVER	67	In	IPADDR	68
DHCPREQUEST; DHCPRELEASE	UDP	DHCP SERVER	67	Out	IPADDR	68

Allowing Client Access to Remote DHCP Server

The following firewall rules allow communication between your DHCP client and a remote server:

```
$INIT or REBINDING: No lease or Lease time expired

$IPTABLES -A OUTPUT -p UDP \
-i $EXTERNAL_INTERFACE \
-s $BROADCAST_0 -sport 68 \
-d $BROADCAST_1 -dport 67 \
-j ACCEPT

$IPTABLES -A INPUT -p UDP \
-i $EXTERNAL_INTERFACE \
-s $BROADCAST_0 -sport 67 \
-d $BROADCAST_1 -dport 68 \
-j ACCEPT
```

```
$IPTABLES -A INPUT -p UDP \
-i $EXTERNAL_INTERFACE \
-s $DHCP_SERVER -sport 67 \
-d BROADCAST_1 -dport 68 \
-j ACCEPT

$IPTABLES -A OUTPUT -p UDP \
-i $EXTERNAL_INTERFACE \
-s $BROADCAST_0 -sport 68 \
-d $BROADCAST_1 -dport 67 \
-j ACCEPT

# As a result of the above, we're supposed to change
# our IP address with this message, which is addressed
# to our new address before the dhcp client has
# received the update.

$IPTABLES -A INPUT -p UDP \
-i $EXTERNAL_INTERFACE \
-s $DHCP_SERVER -sport 67 \
-d $MY_ISP -dport 68 \
-j ACCEPT

$IPTABLES -A INPUT -p UDP \
-i $EXTERNAL_INTERFACE \
-s $DHCP_SERVER -sport 67 \
-d $IPADDR -dport 68 \
-j ACCEPT

$IPTABLES -A INPUT -p UDP \
-i $EXTERNAL_INTERFACE \
-s $BROADCAST_0 -sport 68 \
-d $BROADCAST_1 -dport 67 \
-j ACCEPT

$IPTABLES -A OUTPUT -p UDP \
-i $EXTERNAL_INTERFACE \
-s $IPADDR -sport 68 \
-d $DHCP_SERVER -dport 67 \
-j ACCEPT
```

Notice that DHCP traffic cannot be completely limited to your DHCP server. During initialization sequences, when your client doesn't yet have an assigned IP address or even the server's IP address, packets are broadcast rather than sent point-to-point.

Accessing Remote Network Time Servers (UDP 123)

Network time services such as NTP allow access to one or more public Internet time providers. This is useful to maintain an accurate system clock, particularly if

your internal clock tends to drift, and to establish the correct time and date at boot-up or after a power loss. A small system user should use the service only as a client. Few, if any, small sites have a satellite link to Greenwich, England, a radio link to the United States atomic clock, or an atomic clock of their own lying around.

NTP Protocol

Description	Protocol	Remote Address	Remote Port	In/Out	Local Address	Local Port
Local client request	UDP	timeserver	123	Out	IPADDR	1024:65535
Remote server response	UDP	timeserver	123	In	IPADDR	1024:65535

Allowing Client Access to Remote NTP Servers

```
$IPTABLES -A OUTPUT -p UDP \
-o $EXTERNAL_INTERFACE \
-d <my.time.provider> -dport 123 \
-j ACCEPT

$IPTABLES –A FORWARD –p UDP –m state \
--state ESTABLISHED \
-j ACCEPT
```

Advanced Routing and Traffic Control

IP Command Set

In this section we will present a comprehensive description of the ip utility from Alexey Kuznetsov's IPROUTE2 package. We will start by going through most of the ip command in extreme detail. We will cover the link, addr, route, rule, neigh, tunnel, and monitor parts of the ip command

We will first go through all of the command syntax of the ip command. This is due to the situation, current as of February 2000, that there are no man pages for ip
IP Global Command Syntax

The generic form of the ip command is

```
ip [ OPTIONS ] OBJECT [ COMMAND [ ARGUMENTS ]]
```

OPTIONS:

OPTIONS is a multivalued set of modifiers that affect the general behaviour and output of the ip utility. All options begin with the "-" character and may be used both in long and abbreviated form. Currently the following options are available

-V, -Version --- print the version of the ip utility and exit.

-s, -stats, -statistics --- output more information.

This option may be repeated to increase the verbosity level of the output. As a rule the additional information is device or function statistics or values. In many cases the values output should be considered in the same sense as output from the /proc/ directory where the name of the value is not directly related to the value itself. See later when we run this option with different network device drivers.

-f, -family {inet, inet6, link} --- enforce which protocol family to use.

If this option is not present, the protocol family output to use is guessed from the other command line arguments. If the rest of command line does not provide sufficient information to guess a protocol family, the ip command falls back to a default family of inet in the case of network protocols or to any. Link is a special family identifier meaning that no networking protocol is involved. There are several shortcuts for this option and they are as listed here:

-4 --- shortcut for -family inet.

-6 --- shortcut for -family inet6.

-0 --- shortcut for -family link.

-o, -oneline --- format the output records as single lines by replacing any line feeds with the "\" character.

This option is to provide a convenient method for sending the command output through a pipe. IE: When you want to count the number of output records with wc or you want to to grep through the output. As of 1999-06-30 the IPROUTE2 utility package includes the trivial script rtpr to convert the output back to the original readable form.

-r, -resolve --- use system name resolution to output DNS names

Do not use this option if you are reporting bugs with the ip utility or querying for usage advice. ip itself never uses DNS to resolve names to addresses. This option exists for the administrators convenience only.

OBJECT:

OBJECT is the object type on which you wish to operate on or obtain information about. The object types understood by the current ip utility are link, address, neighbor, route, rule, maddress, mroute, and tunnel.

link --- physical or logical network device.

address --- protocol (IPv4 or IPv6) address on a device.

neighbour --- ARP or NDISC cache entry.

route --- routing table entry.

rule --- rule in routing policy database.

maddress --- multicast address.

mroute --- multicast routing cache entry.

tunnel --- tunnel over IP.

The names of all of the objects may be written in full or abbreviated form. IE: address may be abbreviated as addr or just a. Ho wever if you use these commands within scripts you should make it a habit to always use the full specification of the action. Using the abbreviation makes it easy to use on the command line but hard to understand the logic within scripts. Since you may not be the only person who ever has to deal with your scripts then you should strive to make them as complete as possible.

COMMAND:

COMMAND specifies the action to perform on the object. The set of possible actions depends on the object type. Typically it is possible to add, delete, and show (list) the object(s), but some objects will not allow all of these operations and many have additional actions and commands. Note that the command syntax help which is available for all objects prints out the full list of available commands and argument syntax conventions. If no command is given a default command is assumed. The default command is usually show (list) but if the objects of the class cannot be listed then the default is to print out the command syntax help.

ARGUMENTS:

ARGUMENTS is the list of command options specific to the command. The arguments depend on the command and the object. There are two types of arguments that can be issued:

--- flags - which are abbreviated with a single keyword

--- parameters - consisting of a keyword followed by a value

Each command has a default parameter which is used if the arguments are omitted. IE: The dev parameter is the default for the ip link command thus ip link list eth0 is equivalent to ip link list dev eth0. Within all the command descriptions below we distinguish default parameters with the marker (default).

As we mentioned above for the names of objects, all keywords may be abbreviated with the first or first few unique letters. These shortcuts are convenient when ip is used interactively, but they are not recommended for use in scripts and please do not use them when reporting bugs or asking for help. Officially allowed abbreviations are listed along with the first mention of the command.

Error Conditions

The ip command most commonly fails for the following reasons:

* Wrong command line syntax

This is often due to using an unknown keyword, a wrongly formatted IP address, wrong keyword argument for the command, etc. In this case the ip command exits without performing any actions and prints out an error message containing information about the reason for failure. In some cases it prints out the command syntax help.

* The arguments did not pass self-consistency verification

* ip failed to compile a kernel request from the arguments due to insufficient user provided information

* Kernel returned an error to a syscall. In this case ip prints the error message as it was output from perror(3), prefixed with a comment and the syscall identifier.

* Kernel returned an error to a RTNETLINK request. In this case ip prints the error message as it was output from perror(3) prefixed with "RTNETLINK answers".

Note that all ip command operations are atomic. This means that if the ip command fails it does not change anything in the system. One harmful exception is the ip link command which may change only part of the device parameters given on the command line. We will mention this again in the section on ip link usage and reccomend that all ip link actions be performed individually. This is actually a preferred use for the ip command in general. If you need to perform many repetitions of the command use a script loop or a script as then any generated error messages can be associated with the appropriate ip command action.

It is difficult to list all possible error messages especially the syntax errors. As a rule their meaning should be clear from the context of the command that was issued. For example if we issue the command ip link sub eth0 with the obvious misspelling of set then we get the error message "Command "sub" is unknown, try "ip link help"" which should prompt us to check our command syntax.

In using the ip command there are several facilities that need to be present in order for the command to perform its functions. The ip command talks to the kernel through the NETLINK interface. This is turned on by the NETLINK options which are enabled in the kernel compile. If the ip command does not work or you get an error message then you may not have the needed functions defined or your kernel is not the one you compiled. The most common mistakes are:

* NETLINK is not configured in the kernel. The error message is

"Cannot open netlink socket Invalid value"

* RTNETLINK is not configured in the kernel.

In this case one of the following messages may be printed depending on the actual command issued:

"Cannot talk to rtnetlink Connection refused"

"Cannot send dump request Connection refused"

ip link - network device configuration

A link refers a network device. The ip link object and the corresponding command set allows viewing and manipulating the state of network devices. The commands for the link object are just two, set and show.

ip link set --- change device attributes.

Abbreviations: set, s

Warning

You can request multiple parameter changes with ip link. If you request multiple parameter changes and any ONE change fails then ip aborts immediately after the failure thus the parameter changes previous to the failure have completed and are not backed out on abort. This is the only case where using the ip command can leave your system in an unpredictable state. The solution is to avoid changing multiple parameters with one ip link set call. Use as many individual ip link set commands as necessary to perform the actions you desire.

Arguments:

* dev NAME (default) --- NAME specifies the network device to operate on

* up / down --- change the state of the device to UP or to DOWN

* arp on / arp off --- change NOARP flag status on the device

Note that this operation is not allowed if the device is already in the UP state. Since neither the ip utility nor the kernel check for this condition, you can get very unpredictable results changing the flag while the device is running. It is better to set the device down then issue this command.

* multicast on / multicast off --- change MULTICAST flag on the device.

* dynamic on / dynamic off --- change DYNAMIC flag on the device.

* name NAME --- change name of the device.

Note that this operation is not recommended if the device is running or has some addresses already configured. You can break your systems security and screw up other networking daemons and programs by changing the device name while the device is running or has addressing assigned.

* txqueuelen NUMBER / txqlen NUMBER --- change transmit queue length of the device

* mtu NUMBER --- change MTU of the device.

* address LLADDRESS --- change station address of the interface.

* broadcast LLADDRESS, brd LLADDRESS or peer LLADDRESS --- change link layer broadcast address or peer address in the case of a POINTOPOINT interface

Note that for most physical network devices (Ethernet, TokenRing, etc) changing the link layer broadcast address will break networking. Do not use this argument if you do not understand what this operation really does.

* The ip command does not allow changing the PROMISC or ALLMULTI flags as these flags are considered obsolete and should not be changed administratively.

Examples:

ip link set dummy address 000000000001 --- change station address of the interface dummy.

ip link set dummy up --- start the interface dummy.

ip link show --- look at device attributes.

Abbreviations: show, list, lst, sh, ls, l

Arguments:

* dev NAME (default) --- NAME specifies network device to show.

If this argument is omitted, the command lists all the devices.

* up --- display only running interfaces.

Output:

kuznet@alisa~:$ ip link ls dummy

2: dummy: <BROADCAST,NOARP> mtu 1500 qdisc noop

link/ether 000000000000 brd ffffffffffff

The number followed by a colon is the interface index or ifindex. This number uniquely identifies the interface. If you look at the output from cat /proc/net/dev you will see that the network devices are listed in the same order as the numbering

you see here. After the ifindex is the interface name (eth0, sit0 etc.). The interface name is also unique at any given moment, however interfaces may disappear from the list, such as when the corresponding driver module is unloaded, and another interface with the same name will be created later. Additionally with the ip link set DEVICE name NEWNAME command the system administrator may change the devices name.

The interface name may also have another name or the keyword NONE appended after an "@" sign. This signifies that this device is bound to another device in a master/slave device relationship. Thus packets sent through this device are encapsulated and forwarded on via the master device. If the name is NONE, then the master device is unknown.

After the interface name we see the interface mtu (maximal transfer unit) which determines maximal size of data packet which can be sent as a single packet over this interface.

The qdisc (queuing discipline) shows which queuing algorithm is used on the interface. In particular the keyword noqueue means that this interface does not queue anything and the keyword noop indicates that the interface is in blackhole mode in which all of the packets sent to it are immediately discarded.

The qlen indicates the default transmit queue length of the device measured in packets.

Following all of this inormation is a section within angle brackets. Within the angle brackets is where the interface flags are summarized. The most applicable flags are as follows:

UP --- this device is turned on, ready to accept packets for transmission onto the network and it may receive packets from other nodes on the network.

LOOPBACK --- the interface does not communicate to another hosts. All the packets which are sent through it will be returned back to the sender and nothing but bounced back packets can be received.

BROADCAST --- this device has the facility to send packets to all other hosts sharing the same physical link. Example: Ethernet

POINTOPOINT --- the network has only two ends with two nodes attached. All the packets sent to the link will reach the peer link and all packets received are origined by the peer.

If neither LOOPBACK nor BROADCAST nor POINTOPOINT are set, the interface is assumed to be a NBMA (Non-Broadcast Multi-Access) link. NBMA is the most generic type of device and also the most complicated type of device

because a host attached to a NBMA link cannot send information to any other host without additional manually provided configuration information.

MULTICAST --- an advisory flag noting the interface is aware of multicasting. Broadcasting is particular case of multicasting where the multicast group contains all of the nodes on the link as members. Note that software must NOT interpret the absence of this flag as the incapability of the interface to multicast. Any POINTOPOINT and BROADCAST link is multicasting by definition because we have direct access to all the link neighbours and thus to any particular group of them. The use of high bandwidth multicast transfers is not recommended on broadcast-only networks due to the high expenses associated with the transmission, but such use is not strictly prohibited.

PROMISC --- the device listens and feeds to the kernel all of the traffic on the link. This includes every packet on the network that passes our transceiver. Usually this mode exists only on broadcast links and is used by bridges and network monitoring devices.

ALLMULTI --- the device receives all multicast packets wandering on the link. This mode is used by multicast routers.

NOARP --- this flag is different from the other flags. It has no invariant value and its interpretation depends on network protocols involved. As a rule it indicates that the device does not need any address resolution and that the software or hardware knows how to deliver packets without any help from the protocol stacks.

DYNAMIC --- is an advisory flag marking this interface as dynamically created and destroyed.

SLAVE --- this interface is bonded to other interfaces in order to share link capacities.

Other flags do exist and can be seen in within the angle brackets but they are either obsolete (NOTRAILERS), not implemented (DEBUG), or specific to certain devices (MASTER, AUTOMEDIA and PORTSEL). We will not discuss them here. Additionally the values of the PROMISC and ALLMULTI flags as shown by the ifconfig utility and by the ip utility are different. The ip link list command provides the current true device state, whereas ifconfig shows the flag state which was set through ifconfig itself.

The second line of the output from the example contains information about the link layer addresses associated with the device. The first word (ether, sit) defines the interface hardware type which then determines the format and semantics of the addresses and thus logically is part of the address itself. The default format of station and broadcast addresses (or peer addresses for pointopoint links) is a sequence of hexadecimal bytes separated by colons. However some link types may

instead have their own natural address formats which are used in the presentation. IE: The addresses of IP tunnels are printed as dotted-quad IP addresses. While NBMA links have no well-defined broadcast or peer address, this field may contain useful information such as the address of a broadcast relay or the address of an ARP server. Multicast addresses are not shown by this command, see ip maddr list output.

When given the option -statistics ip will print the interface statistics as additional information in the listing. Note that you can give this option multiple times with each repetition increasing the verbosity of output.

kuznet@alisa~ $ ip -s link ls eth0

3: eth0: <BROADCAST,MULTICAST,UP> mtu 1500 qdisc cbq qlen 100

link/ether 00a0cc661878 brd ffffffffffff

RX bytes packets errors dropped overrun mcast

2449949362 2786187 0 0 0 0

TX bytes packets errors dropped carrier collsns

178558497 1783945 332 0 332 35172

The RX and TX lines summarize receiver and transmitter statistics. The information output breaks down into:

bytes --- total number of bytes received or transmitted on the interface.

This number wraps when the maximal length of the natural data type on the architecture is exceeded. In order to provide correct long term data from this output these statistics should be continuously monitored. Continuous monitoring of this data requires a user level daemon to sample the output periodically.

packets --- total number of packets received or transmitted on the interface.

errors --- total number of receiver or transmitter errors.

dropped --- total number of packets dropped because of lack of resources.

overrun --- total number of receiver overruns resulting in packet drops. As a rule if the interface is overrun you have a serious problem either within the kernel or your machine is too slow to handle the speed of this interface.

mcast --- total number of received multicast packets. This option is supported only on certain devices.

carrier --- total number of link media failures such as those due to lost carrier.

collsns --- total number of collision events on Ethernet-like media. This number has different interpretations on other link types.

compressed --- total number of compressed packets. It is available only for links using VJ header compression.

When you issue the -statistics option more than once you get additional output depending on the statistics supported by the device itself as in the following example with Ethernet:

ip address - protocol address management

Abbreviations: address, addr, a

Arguments: add, delete, flush, show (list)

The address refers to a protocol (IP or IPv6) address attached to a network device. Each device must have at least one address in order to use the corresponding protocol. It is possible to have several different addresses attached to one device. These addresses are not discriminated within the protocol structure so that the term alias is not quite appropriate for such multiple addresses and we will not refer to this situation in those terms.

The ip addr command allows you to look at the addresses and their properties on an interface. You can add new addresses and delete old ones without regard to any ordering. Later on we will discuss the concept of primary and secondary addresses as applied to Linux.

ip address add --- add new protocol address.

Abbreviations: add, a

Arguments:

dev NAME --- name of the device to which we add the address

local ADDRESS (default) --- address of the interface.

The format of the address depends on the protocol. IPv4 uses dotted quad and IPv6 uses a sequence of hexadecimal halfwords separated by colons. The ADDRESS

may be followed by a slash and a decimal number, which encodes network prefix (netmask) length in CIDR notation. If no CIDR netmask notation is specified then the command assumes a host (/32 mask) address is specified.

peer ADDRESS--- address of remote endpoint for pointopoint interfaces. Again, the ADDRESS may be followed by a slash and decimal number, encoding the network prefix length. If a peer address is specified then the local address cannot have a network prefix length as the network prefix is associated with the peer rather than with the local address. In other words, netmasks can only be assigned to peer addresses when specifying both peer and local addresses.

broadcast ADDRESS --- broadcast address on the interface.

The special symbols "+" and "-" can be used instead of specifying the broadcast address. In this case the broadcast address is derived by either setting all of the interface host bits to one (+) or by setting all of the interface host bits to zero (-). In most modern implementations of IPv4 networking you will want to use the (+) setting. See the ipup init script in Chapter 15. Unlike ifconfig, the ip command does not set a broadcast address unless explicitly requested.

label NAME --- Each address may be tagged with a label string.

In order to preserve compatibility with Linux-2.0 net aliases, this string must coincide with the name of the device or must be prefixed with device name followed by a colon. (eth0:duh)

scope SCOPE_VALUE --- scope of the area within which this address is valid.

The available scopes are listed in the file

/etc/iproute2/rt_scopes. The predefined scope values are:

global --- the address is globally valid.

site --- (IPv6 only) address is site local, valid only inside this site.

link --- the address is link local, valid only on this device.

host --- the address is valid only inside this host.

Examples:

ip addr add 127.0.0.1/8 dev lo brd + scope host

--- adds the usual loopback address to loopback device. The device must be enabled before this address will show up.

ip addr add 10.0.0.1/24 brd + dev eth0

--- adds address 10.0.0.1 with prefix length 24 (netmask 255.255.255.0) and standard broadcast to interface eth0

ip address delete --- delete protocol address.

Abbreviations: delete, del, d

Arguments:

The arguments coincide with arguments of ip addr add. The device name is a required argument, the rest are optional. If no arguments are given, the first address listed is deleted.

Examples:

ip addr del 127.0.0.1/8 dev lo

--- deletes the loopback address from loopback device.

Alexey states:

"It would be better not to try to repeat this experiment 8-}"

Delete all IPv4 addresses on interface eth0:

while ip -f inet addr del dev eth0; do

nothing

done

Another method to disable IP on an interface using ip addr flush is discussed later.

ip address show --- look at protocol addresses.

Abbreviations: show, list, lst, sh, ls, l

Arguments:

dev NAME (default) --- name of the device.

scope SCOPE_VAL --- list only addresses with this scope.

to PREFIX --- list only addresses matching this prefix.

label PATTERN --- list only addresses with labels matching the PATTERN.

PATTERN is the usual shell regexp style pattern.

dynamic / permanent --- (IPv6 only) list only addresses installed due to stateless address configuration or list only the permanent (not dynamic) addresses.

tentative --- (IPv6 only) list only addresses, which did not pass duplicate address detection.

deprecated --- (IPv6 only) list only deprecated addresses.

primary / secondary --- list only primary (or secondary) addresses.

Example:

kuznet@alisa~ $ ip addr ls eth0

3: eth0: <BROADCAST,MULTICAST,UP> mtu 1500 qdisc cbq qlen 100

link/ether 00a0cc661878 brd ffffffffffff

inet 193.233.7.90/24 brd 193.233.7.255 scope global eth0

inet6 3ffe2400012a0ccfffe661878/64 scope global dynamic

valid_lft forever preferred_lft 604746sec

inet6 fe802a0ccfffe661878/10 scope link

The first two lines coincide with the output of ip link list as it is only natural to interpret link layer addresses as being addresses of the protocol family AF_PACKET. The list of IPv4 and IPv6 addresses follows accompanied by additional attributes such as scope value, flags, and address label. Address flags are set by the kernel and cannot be changed administratively. Currently the following flags are defined:

secondary --- this address is not used when selecting the default source address for outgoing packets. An IP address becomes secondary if another address within the same prefix (network) already exists. The first address within the prefix is primary and is the tag address for the group of all the secondary addresses. When the primary address is deleted all of the secondaries are purged too. See the examples for the actual functionality of these steps.

dynamic --- the address was created due to stateless autoconfiguration. In this case the output also contains information on the times for which the address remains valid. After the preferred lifetime (preferred_lft) expires the address is moved to the deprecated state and after the valid lifetime (valid_lft) expires the address is finally invalidated.

deprecated --- the address is deprecated. It is still valid but cannot be used by newly created connections. See dynamic above.

tentative --- the address is not used because duplicate address detection is still not complete or has failed.

IP Interface Primary and Secondary Addressing:

To explain the actual relationship between primary and secondary addresses we will run the following experiment.

ip addr add 10.1.1.1/24 dev dummy

ip addr add 10.1.1.2/24 dev dummy

Now look at the output:

ip addr list dummy

3: dummy: <BROADCAST,MULTICAST,NOARP> mtu 1500 qdisc noop

link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff

inet 10.1.1.1/24 scope global dummy

inet 10.1.1.2/24 scope global secondary dummy

Now add in some addresses still in that network but add them as host addresses:

ip addr add 10.1.1.3/32 dev dummy

ip addr add 10.1.1.4/25 dev dummy

And run our list command:

ip addr list dummy

ip address flush --- flush protocol addresses.

Abbreviations: flush, f

Arguments:

This commands flushes protocol addresses selected by some criteria. This command has the same arguments as show. The major difference is that this command will not run if no arguments are given. Otherwise you could delete all of your addresses by mistake. This command (and the other flush commands described below) are very dangerous. If you make a mistake the command does not ask or forgive but really will creully purge all of your addresses. Be warned!

With the option -statistics the command becomes verbose and prints out the number of deleted addresses and number of processing rounds made in order to flush the address list. If the -statistics option is given twice then ip addr flush also dumps all of the deleted addresses in the full format as described in the ip addr list section.

Examples:

Delete all the addresses from private network 10.0.0.0/8:

netadm@amber~ # ip -stat -stat addr flush to 10/8

Another instructive example is deleting all IPv4 addresses from all Ethernet interfaces in the system:

netadm@amber~ # ip -4 addr flush label "eth*"

And the last example shows how to flush all the IPv6 addresses acquired by the host from stateless address autoconfiguration after enabling forwarding or disabling autoconfiguration.

netadm@amber~ # ip -6 addr flush dynamic

ip neighbour --- neighbour/arp table management.

Abbreviations: neighbour, neighbor, neigh, n

The neighbour table objects establish bindings between protocol addresses and link layer addresses for hosts sharing the same physical link. Neighbour object entries are organized into tables. The IPv4 neighbour object table is known under another name as the ARP table. These commands allow you to look at the neighbour table bindings and their properties, to add new neighbour table entries, and to delete old ones.

Arguments:

add, change, replace, delete, flush and show (list)

ip neighbour add --- add new neighbour entry

ip neighbour change --- change existing entry

ip neighbour replace --- add new or change existing entry

add, a; change, chg; replace, repl

These commands create new neighbour records or update existing ones.

to ADDRESS (default) --- protocol address of the neighbour. It is either an IPv4 or IPv6 address.

dev NAME --- the interface to which this neighbour is attached

lladdr LLADDRESS --- link layer address of the neighbour. LLADDRESS can be null.

nud NUD_STATE --- state of the neighbour entry. nud is an abbreviation for "Neighbour Unreachability Detection". This state can take one of the following values:

permanent --- the neighbour entry is valid forever and can be removed only administratively.

noarp --- the neighbour entry is valid, no attempts to validate this entry will be made but it can be removed when its lifetime expires.

reachable --- the neighbour entry is valid until reachability timeout expires.

stale --- the neighbour entry is valid, but suspicious. This option to ip neighbour does not change the neighbour state if the entry was valid and the address has not been changed by this command.

Examples:

ip neigh add 10.0.0.3 lladdr 000001 dev eth0 nud perm

--- add permanent ARP entry for neighbour 10.0.0.3 on the device eth0.

ip neigh chg 10.0.0.3 dev eth0 nud reachable

--- change its state to reachable.

ip neighbour delete --- delete neighbour entry.

Abbreviations: delete, del, d.

This command invalidates a neighbour entry.

The arguments are the same as with ip neigh add, only lladdr and nud are ignored.

Example:

ip neigh del 10.0.0.3 dev eth0

--- invalidate ARP entry for neighbour 10.0.0.3 on the device eth0.

Deleted neighbour entry will not disappear from the tables immediately; if it is in use it cannot be deleted until the last client will release it, otherwise it will be destroyed during the next garbage collection.

WARNING!

Attempts to delete or to change manually a noarp entry created by kernel may result in unpredictable behaviour. More specifically the kernel may start trying to resolve this address even on NOARP interfaces or change the address to multicast or broadcast.

ip neighbour show --- list neighbour entries.

Abbreviations: show, list, sh, ls.

This commands displays neighbour tables.

Arguments:

to ADDRESS (default) --- prefix selecting neighbours to list.

dev NAME --- list only neighbours attached to this device.

unused --- list only neighbours, which are not in use now.

nud NUD_STATE --- list only neighbour entries in this state. NUD_STATE takes values listed below after the example or the special value all, which means all the states. This option may occur more than once. If this option is absent, ip lists all the entries except for none and noarp.

Example:

kuznet@alisa~ $ ip neigh ls

dev lo lladdr 000000000000 nud noarp

fe80200cfffe763f85 dev eth0 lladdr 00000c763f85 router nud stale

0.0.0.0 dev lo lladdr 000000000000 nud noarp

193.233.7.254 dev eth0 lladdr 00000c763f85 nud reachable

193.233.7.85 dev eth0 lladdr 00e01e633900 nud stale

kuznet@alisa~ $

The first word of each line is the protocol address of the neighbour followed by the device name. The rest of the line describes the contents of neighbour entry identified by the pair (device, address).

lladdr is link layer address of the neighbour.

nud is the state of ``neighbour unreachability detection for this entry. The full list of the possible nud states with minimal descriptions are:

none --- state of the neighbour is void.

incomplete --- the neighbour is in process of resolution.

reachable --- the neighbour is valid and apparently reachable.

stale --- the neighbour is valid, but probably it is already unreachable, so that kernel will try to check it at the first transmission.

delay --- a packet has been sent to the stale neighbour, kernel waits for confirmation.

probe --- delay timer expired, but no confirmation was received. Kernel has started to probe neighbour with ARP/NDISC messages.

failed --- resolution has failed.

noarp --- the neighbour is valid, no attempts to check the entry will be made.

permanent --- it is noarp entry, but only administrator may remove the entry from neighbour table.

Link layer address is valid in all the states except for none, failed and incomplete.

IPv6 neighbours can be marked with the additional flag router, which means that that neighbour introduced itself as an IPv6 router.

Option -statistics provides some usage statistics,

kuznet@alisa~ $ ip -s n ls 193.233.7.254

193.233.7.254 dev eth0 lladdr 00000c763f85 ref 5 used 12/13/20 \

nud reachable

kuznet@alisa~ $

Here ref is number of users of this entry, and used is a triplet of time intervals in seconds separated by slashes. The triplet of numbers is coded as {used/confirmed/updated}. In this example they show that

The entry was used 12 seconds ago.

The entry was confirmed 13 seconds ago.

The entry was updated 20 seconds ago.

ip neighbour flush --- flush neighbour entries.

Abbreviations: flush, f.

This commands flushes the neighbour tables. Entries may be selected to flush by various criteria.

This command has the same arguments as show. Note that it will not run when no arguments are given, and that the default neighbour states to be flushed do not include permanent or noarp.

With the option -statistics the command becomes verbose and prints out the number of deleted neighbours and number of rounds made in flushing the neighbour table. If the option is given twice, ip neigh flush also dumps all the deleted neighbours in the format described in the previous subsection.

netadm@alisa~ # ip -s -s n f 193.233.7.254

193.233.7.254 dev eth0 lladdr 00000c763f85 ref 5 used 12/13/20 \

nud reachable

Round 1, deleting 1 entries

Flush is complete after 1 round

ip route - routing table management.

Abbreviations: route, ro, r.

This command manages the route entries within the kernel routing tables. The kernel routing tables keep information about protocol paths to other networked nodes.

Each route entry has a key consisting of the protocol prefix, which is the pairing of the network address and network mask length, and optionally the Type of Service (TOS) value. An IP packet matches to the route if the highest bits of the packets destination address are equal to the route prefix at least up to the prefix length and if the TOS of the route is zero or equal to TOS of the packet.

If several routes match to the packet, the following pruning rules are used to select the best one:

1. The longest matching prefix is selected, all shorter ones are dropped.

2. If the TOS of some route with the longest prefix is equal to TOS of the packet then routes with different TOS are dropped.

3. If no exact TOS match was found and routes with TOS=0 exist, the rest of the routes are pruned. Otherwise the route lookup fails.

4. If several routes remain after steps 1-4 have been tried then routes with the best preference value are selected.

5. If we still have several routes then the first of them is selected.

Note the ambiguity of action 5. Unfortunately, Linux historically allowed such a bizarre situation. The sense of the word "the first" depends on the literal order in which the routes were added to the routing table and it is practically impossible to maintain a bundle of such routes in any such order.

For simplicity we will limit ourselves to the case wherein such a situation is impossible and routes are uniquely identified by the triplet of {prefix, tos, preference}. Using the ip command for route creation and manipulation makes it impossible to create such non-unique routes.

One useful exception to this rule is the default route on non-forwarding hosts. It is "officially" allowed to have several fallback routes in cases when several routers are present on directly connected networks. In this case Linux performs "dead gateway detection" as controlled by neighbour unreachability detection and references from the transport protocols to select the working router thus the ordering of the routes is not essential. However in this specific case it is not recommended that you manually fiddle with default routes but instead use the Router Discovery protocol. Actually Linux IPv6 does not even allow user level applications access to default routes.

Of course the route selection steps above are not performed in exactly this sequence. The routing table in the kernel is kept in a data structure which allows achieving the final result with minimal cost. Without depending on any particular routing algorithm implemented in the kernel we can summarize the sequence above as: Route is identified by triplet {prefix,tos,preference} key which uniquely locates the route in the routing table.

Route attributes: Each route key refers to a routing information record. The routing information record contains the data required to deliver IP packets, such as output device and next hop router, and additional optional attributes, such as path MTU or the preferred source address for communicating to that destination.

Route types: It is important that the set of required and optional attributes depends on the route type. The most important route type is a unicast route which describes real paths to another hosts. As a general rule, common routing tables only contain unicast routes. However other route types with different semantics do exist. The full list of types understood by the Linux 2.2 kernel is:

unicast --- the route entry describes real paths to the destinations covered by route prefix.

unreachable --- these destinations are unreachable; packets are discarded and the ICMP message host unreachable (ICMP Type 3 Code 1) is generated. The local senders get error EHOSTUNREACH.

blackhole --- these destinations are unreachable; packets are silently discarded. The local senders get error EINVAL.

prohibit --- these destinations are unreachable; packets are discarded and the ICMP message communication administratively prohibited (ICMP Type 3 Code 13) is generated. The local senders get error EACCES.

local --- the destinations are assigned to this host, the packets are looped back and delivered locally.

broadcast --- the destinations are broadcast addresses, the packets are sent as link broadcasts.

throw --- special control route used together with policy rules. If a throw route is selected then lookup in this particular table is terminated pretending that no route was found. Without any policy routing it is equivalent to the absence of the route in the routing table, the packets are dropped and ICMP message net unreachable (ICMP Type 3 Code 0) is generated. The local senders get error ENETUNREACH.

nat --- special NAT route. Destinations covered by the prefix are considered as dummy (or external) addresses, which require translation to real (or internal) ones before forwarding. The addresses to translate to are selected with the attribute via.

anycast --- (not implemented) the destinations are anycast addresses assigned to this host. They are mainly equivalent to local addresses with the difference that such addresses are invalid to be used as the source address of any packet.

multicast --- special type, used for multicast routing. It does not present in normal routing tables.

Route tables: Linux can place routes within multiple routing tables identified by a number in the range from 1 to 255 or by a name taken from the file /etc/iproute2/rt_tables. By default all normal routes are inserted to the table main (ID 254) and the kernel uses only this table when calculating routes.

Actually another routing table always exists which is invisible but even more important. It is the local table (ID 255). This table consists of routes for local and broadcast addresses. The kernel maintains this table automatically and administrators should not modify it and do not even need to look at it in normal operation.

The multiple routing tables come into play when policy routing is used. In policy routing the routing table identifier becomes effectively one more parameter added to the key triplet {prefix,tos,preference}. Thus under policy routing the route is obtained by {tableid,key triplet} identifing the route uniquely. So you can have

several identical routes in different tables that will not conflict as we had mentioned above in in the description of "the first" mechanism.

ip route add --- add new route

ip route change --- change route

ip route replace --- change route or add new one.

Abbreviations: add, a; change, chg; replace, repl.

Arguments:

to PREFIX or to TYPE PREFIX (default) --- destination prefix of the route. If TYPE is omitted, ip assumes type unicast. Another values of TYPE are listed above. PREFIX is IPv4 or IPv6 address optionally followed by slash and prefix length. If the length of the prefix is missing, ip assumes full-length host route. Also there is one special PREFIX --- default --- which is equivalent to IP 0/0 or to IPv6 /0.

tos TOS or dsfield TOS --- Type Of Service (TOS) key. This key has no mask associated and the longest match is understood as first, compare TOS of the route and of the packet, if they are not equal, then the packet still may match to a route with zero TOS. TOS is either 8bit hexadecimal number or an identifier from /etc/iproute2/rt_dsfield.

metric NUMBER or preference NUMBER --- preference value of the route. NUMBER is an arbitrary 32bit number.

table TABLEID --- table to add this route. TABLEID may be a number or a string from the file /etc/iproute2/rt_tables. If this parameter is omitted, ip assumes table main, with exception of local, broadcast and nat routes, which are put to table local by default.

dev NAME --- the output device name.

via ADDRESS --- the address of nexthop router. Actually, the sense of this field depends on route type. For normal unicast routes it is either true nexthop router or, if it is a direct route installed in BSD compatibility mode, it can be a local address of the interface. For NAT routes it is the first address of block of translated IP destinations.

src ADDRESS --- the source address to prefer using when sending to the destinations covered by route prefix. This address must be defined on a local machine interface. This will come into play when routes and rules are combined with the masquerade rules of the ipchains firewall we discuss later.

realm REALMID --- the realm which this route is assigned to. REALMID may be a number or a string from the file /etc/iproute2/rt_realms.

mtu MTU or mtu lock MTU --- the MTU along the path to destination. If modifier lock is not used, MTU may be updated by the kernel due to Path MTU Discovery. If the modifier lock is used then no path MTU discovery will be performed and all the packets will be sent without the DF bit set for the IPv4 case or fragmented to the MTU for the IPv6 case.

window NUMBER --- the maximal advertised window for TCP to these destinations measured in bytes. This parameter limits the maximal data bursts our TCP peers are allowed to send to us.

rtt NUMBER --- the initial RTT (``Round Trip Time) estimate. Actually, in Linux 2.2 and 2.0 it is not RTT but the initial TCP retransmission timeout. The kernel forgets it as soon as it receives the first valid ACK from peer. Alas, this means that this attribute affects only the connection retry rate and is hence useless.

nexthop NEXTHOP --- nexthop of multipath route. NEXTHOP is a complex value with its own syntax as follows:

via ADDRESS is nexthop router.

dev NAME is output device.

weight NUMBER is weight of this element of multipath route

reflecting its relative bandwidth or quality.

scope SCOPE_VAL --- scope of the destinations covered by the route prefix. SCOPE_VAL may be a number or a string from the file /etc/iproute2/rt_scopes. If this parameter is omitted, ip assumes scope global for all gatewayed unicast routes, scope link for direct unicast routes and broadcasts and scope host for local routes.

protocol RTPROTO --- routing protocol identifier of this route. RTPROTO may be a number or a string from the file /etc/iproute2/rt_protos. If the routing protocol ID is not given ip assumes the protocol is boot. IE. This route has been added by someone who does not understand what they are doing. Several of these protocol values have a fixed interpretation.

redirect --- route was installed due to ICMP redirect.

kernel --- route was installed by the kernel during autoconfiguration.

boot --- route was installed during bootup sequence. If a routing daemon will start, it will purge all of them. This is the value assigned to manually inserted routes that do not have a protocol specified.

static --- route was installed by administrator to override dynamic routing. Routing daemon(s) will respect them and advertise them if it is so configured.

ra --- route was installed by Router Discovery protocol.

Note that the rest of values are not reserved and administrator is free to assign or not assign protocol tags. Routing daemons at least should take care of setting some unique protocol values for themselves such as they are assigned in rtnetlink.h or in the rt_protos database.

onlink --- pretend that the nexthop is directly attached to this link, even if it does match any interface prefix. One application of this option may be found in ip tunnels between dissimilar addresses.

equalize --- allow packet by packet randomization on multipath routes. Without this modifier route will be frozen to one selected nexthop, so that load splitting will occur only on per-flow base. Equalize works only if the appropriate kernel configuration option is chosen or if the kernel is patched.

Two more commands, prepend and append do exist. Prepend does the same thing as the classic route add command by adding the route even if another route to the same destination already exists. The opposite case is append which adds the route to the end of the list. We strongly reccommend that you avoid using these commands.

Unfortunately, IPv6 currently only understands the append command correctly, all the rest of the set translating to append. Certainly, this will change in the future.

Examples:

Add a plain route to network 10.0.0/24 via gateway 193.233.7.65

ip route add 10.0.0/24 via 193.233.7.65

change it to a direct route via device dummy

ip ro chg 10.0.0/24 via 193.233.7.65 dev dummy

Add default multipath route splitting load between ppp0 and ppp1

ip route add default scope global nexthop dev ppp0 nexthop dev ppp1

Note the scope value which is not necessary but prompts the kernel that this route is gatewayed rather than direct. Actually, if you know the addresses of the remote endpoints it would be better to specify them using the parameter via.

NAT the address 192.203.80.144 to 193.233.7.83 before forwarding

ip route add nat 192.203.80.142 via 193.233.7.83

Note that the reverse NAT translation is setup with policy rules as described in the policy routing section.

ip route delete

Abbreviations: delete, del, d.

ip route del has the same arguments as ip route add but their semantics are a bit different.

Key values (dest, tos, preference and table) select the route to delete. If any optional attributes are present, ip verifies that they coincide with attributes of the route to delete. If no route with given key and attributes is found then ip route del fails.

Linux kernel 2.0 had the ability to delete a route selected only by the prefix address while ignoring its netmask. This option does not exist anymore due to the ambiguous nature of the selection. If you wish to have such functionality then look at the ip route flush command which provides a richer set of capabilities.

Examples:

Delete the multipath route created by the add example previously

ip route del default scope global nexthop dev ppp0 nexthop dev ppp1

ip route show

Abbreviations: show, list, sh, ls, l.

This format of the command allows viewing the routing tables contents and looking at route(s) as selected by some criteria.

Arguments:

to SELECTOR (default) --- select routes only from the given range of destinations. SELECTOR has optional modifiers (root, match or exact) and a prefix.

root PREFIX selects routes with prefixes not shorter than PREFIX. IE: root 0/0 selects all the routing table.

match PREFIX selects routes with prefixes not longer than PREFIX. match 10.0/16 selects 10.0/16, 10/8 and 0/0, but it does not select 10.1/16 and 10.0.0/24.

exact PREFIX (or just PREFIX) selects routes exactly with this prefix.

If none of these options are present then the ip command assumes root 0/0 which lists the entire table.

tos TOS or dsfield TOS --- Select only routes with given TOS.

table TABLEID --- Show routes from this table(s). Default setting is to show table main (ID 254). TABLEID may be either ID of a real table or one of the special values:

all --- list all the tables.

cache --- dump routing cache.

IPv6 has only a single table, however splitting into main, local, and cache is emulated by the ip utility.

cloned or cached --- list cloned routes which are routes dynamically forked off of other routes because some route attribute (like MTU) was updated. It is equivalent to table cache.

from SELECTOR --- the same syntax as to SELECTOR but bounds the source address range rather than the destination. Note that the from option only works with cloned routes.

protocol RTPROTO --- list only routes of this protocol.

scope SCOPE_VAL --- list only routes with this scope.

type TYPE --- list only routes of this type.

dev NAME --- list only routes going via this device.

via PREFIX --- list only routes going via selected by PREFIX nexthop routers.

src PREFIX --- list only routes with preferred source addresses selected by PREFIX.

realm REALMID or realms FROMREALM/TOREALM --- list only routes with these realms.

Using this command is best explained by running through some examples.

Example: Let us count the routes of protocol gated/bgp on a router

kuznet@amber~ $ ip route list proto gated/bgp | wc

1413 9891 79010

kuznet@amber~ $

To count size of routing cache we have to use option -o, because cached attributes can take more than one line of the output

kuznet@amber~ $ ip -o route list cloned | wc

159 2543 18707

kuznet@amber~ $

The output of this command consists of per route records separated by line feeds. However, some records may consist of more than one line particularly when the route is cloned or you have requested additional statistics. If the option -o is given, then line feeds separating lines inside records are replaced with backslash sign.

The output has the same syntax as arguments given to ip route add, so that it can be understood easily.

kuznet@amber~ $ ip route list 193.233.7/24

193.233.7.0/24 dev eth0 proto gated/conn scope link \

src 193.233.7.65 realms inr.ac

kuznet@amber~ $

If you list cloned entries the output contains other attributes, which are evaluated during route calculation and updated during route lifetime. The example of the output is:

kuznet@amber~ $ ip route list 193.233.7.82 table cache

193.233.7.82 from 193.233.7.82 dev eth0 src 193.233.7.65 \

realms inr.ac/inr.ac

cache <src-direct,redirect> mtu 1500 rtt 300 iif eth0

193.233.7.82 dev eth0 src 193.233.7.65 realms inr.ac

cache mtu 1500 rtt 300

kuznet@amber~ $

This route looks a bit strange, does it not? Did you notice that this is the path from 193.233.7.82 back to 193.233.82? In the section on ip route get you will see how this route is created.

The second line which starts with the word cache shows the additional attributes which normal routes do not possess. The cache flags contained within the angle brackets are:

local --- packets are delivered locally. It stands for loopback unicast routes, for broadcast routes, and for multicast routes if this host is member of the corresponding group.

reject --- the path is bad. Any attempt to use it results in error. See attribute error below.

mc --- the destination is multicast.

brd --- the destination is broadcast.

src-direct --- the source is on a directly connected interface.

redirected --- the route was created by an ICMP Redirect.

redirect --- packets going via this route will trigger ICMP redirect.

fastroute --- route is eligible to be used for fastroute.

equalize --- make packet by packet randomization along this path.

dst-nat --- destination address requires translation.

src-nat --- source address requires translation.

masq --- source address requires masquerading.

notify --- (not implemented) change/deletion of this route will trigger RTNETLINK notification.

The following are optional attributes that may be present:

error --- on reject routes this is the error code returned to local senders when they try to use this route. These error codes are translated to ICMP error codes sent to remote senders according to the rules described above in the subsection devoted to route types.

expires --- this entry will expire after this timeout.

iif --- the packets for this path are expected to arrive on this interface.

Giving the option -statistics will show further information about this route:

users --- number of users of this entry.

age --- shows when this route was used last time.

used --- number of lookups of this route since its creation.

ip route flush - allows group deletion of routes

Abbreviations: flush, f.

This command allows flushing routes as selected by some criteria.

The arguments have the same syntax and semantics as the arguments of ip route show but the routing tables are purged rather than listed. The only difference is the default action performed. Where the ip route show command dumps the main IP routing table, ip route flush prints the help page. The reason for this difference does not require an explanation does it?

With the option -statistics the command becomes verbose and prints out the number of deleted routes and the number of rounds needed to flush the routing table. If the option is given twice then ip route flush also dumps all deleted routes in the format described in the previous subsection.

Examples:

The first example flushes all the gatewayed routes from main table such as after a routing daemon crash.

netadm@amber~ # ip -4 ro flush scope global type unicast

This option deserved to be put into the scriptlet routef available within the IPROUTE2 utility distribution. This option was described in the route(8) man page as borrowed from BSD but was never implemented in Linux.

The second example is flushing all IPv6 cloned routes:

netadm@amber~ # ip -6 -s -s ro flush cache

3ffe2400220affffef4c5d1 via 3ffe2400220affffef4c5d1 \

dev eth0 metric 0

cache used 2 age 12sec mtu 1500 rtt 300

3ffe2400280adfffeb78034 via 3ffe2400280adfffeb78034 \

dev eth0 metric 0

cache used 2 age 15sec mtu 1500 rtt 300

3ffe2400280c8fffe595bcc via 3ffe2400280c8fffe595bcc \

dev eth0 metric 0

cache users 1 used 1 age 23sec mtu 1500 rtt 300

3ffe2400012a0ccfffe661878 via 3ffe2400012a0ccfffe661878 \

dev eth1 metric 0

cache used 2 age 20sec mtu 1500 rtt 300

3ffe240001a0020fffe71fb30 via 3ffe240001a0020fffe71fb30 \

dev eth1 metric 0

cache used 2 age 33sec mtu 1500 rtt 300

ff021 via ff021 dev eth1 metric 0

cache users 1 used 1 age 45sec mtu 1500 rtt 300

Round 1, deleting 6 entries

Flush is complete after 1 round

netadm@amber~ # ip -6 -s -s ro flush cache

Nothing to flush.

The third example is flushing BGP routing tables after gated death.

netadm@amber~ # ip ro ls proto gated/bgp wc

1408 9856 78730

netadm@amber~ # ip -s ro f proto gated/bgp

Round 1, deleting 1408 entries

Flush is complete after 1 round

netadm@amber~ # ip ro f proto gated/bgp

Nothing to flush.

netadm@amber~ # ip ro ls proto gated/bgp

ip route get - obtain route pathing

Abbreviations: get, g.

This command gets a single route to a destination and prints its contents exactly as kernel sees it.

Arguments:

to ADDRESS (default) --- destination address.

from ADDRESS --- source address.

tos TOS or dsfield TOS --- Type Of Service.

iif NAME --- device, which this packet is expected to arrive from.

oif NAME --- enforce output device, which this packet will be routed out.

connected --- if no source address (option from) was given, relookup the route with the source address set to the preferred address as received from the first lookup. If policy routing is used this may be a different route.

Note that this operation is not equivalent to ip route show. ip route show shows the existing routes, ip route get resolves them and creates new clones if necessary. Essentially, ip route get is equivalent to actually sending a packet along this path. If the argument iif is not given then the kernel creates a route to output packets towards requested destination. This is equivalent to pinging the destination then running ip route list cache but in the case of ip route get no packets are actually sent. With the argument iif present the kernel pretends that a packet has arrived from this interface and searches for a path to forward the packet. This command outputs routes in the same format as ip route ls.

Examples:

Find route to output packets to 193.233.7.82:

kuznet@amber~ $ ip route get 193.233.7.82

193.233.7.82 dev eth0 src 193.233.7.65 realms inr.ac

cache mtu 1500 rtt 300

kuznet@amber~ $

Find route to forward packets arriving on eth0 from 193.233.7.82 and destined to 193.233.7.82:

kuznet@amber~ $ ip route get 193.233.7.82 from 193.233.7.82 iif eth0

193.233.7.82 from 193.233.7.82 dev eth0 src 193.233.7.65 \

realms inr.ac/inr.ac

cache <src-direct,redirect> mtu 1500 rtt 300 iif eth0

kuznet@amber~ $

This is the operation that created the funny route in the examples to ip route list with 193.233.7.82 looped back to 193.233.7.82. Note the redirect flag present on the output.

Find multicast route for packets arriving on eth0 from host 193.233.7.82 and destined to multicast group 224.2.127.254 assuming that a multicast routing daemon is running such as in this case we are running pimd.

kuznet@amber~ $ ip route get 224.2.127.254 from 193.233.7.82 iif eth0

multicast 224.2.127.254 from 193.233.7.82 dev lo \

src 193.233.7.65 realms inr.ac/cosmos

cache <mc> iif eth0 Oifs eth1 pimreg

kuznet@amber~ $

This route differs from the ones seen before. It contains a normal part and a multicast part. The normal part is used to deliver or not deliver the packet to local IP listeners. In this case the router is not acting as a member of the multicast group so the route has no local flag and only forwards packets. The output device for such entries is always loopback. The multicast part consists of an additional Oifs list showing the output interfaces.

Now it is time for a more complicated example. Let us add an invalid gatewayed route for a destination which is really directly connected.

netadm@alisa~ # ip route add 193.233.7.98 via 193.233.7.254

netadm@alisa~ # ip route get 193.233.7.98

193.233.7.98 via 193.233.7.254 dev eth0 src 193.233.7.90

cache mtu 1500 rtt 3072

and probe it with ping

netadm@alisa~ # ping -n 193.233.7.98

PING 193.233.7.98 (193.233.7.98) from 193.233.7.90 56 data bytes

From 193.233.7.254 Redirect Host(New nexthop 193.233.7.98)

64 bytes from 193.233.7.98 icmp_seq=0 ttl=255 time=3.5 ms

From 193.233.7.254 Redirect Host(New nexthop 193.233.7.98)

64 bytes from 193.233.7.98 icmp_seq=1 ttl=255 time=2.2 ms

64 bytes from 193.233.7.98 icmp_seq=2 ttl=255 time=0.4 ms

64 bytes from 193.233.7.98 icmp_seq=3 ttl=255 time=0.4 ms

64 bytes from 193.233.7.98 icmp_seq=4 ttl=255 time=0.4 ms

^C

--- 193.233.7.98 ping statistics ---

5 packets transmitted, 5 packets received, 0% packet loss

round-trip min/avg/max = 0.4/1.3/3.5 ms

What occured? The router at 193.233.7.254 understood that we have a much better path to the destination and sent us an ICMP redirect message. We now retry ip route get to see what we have in our routing tables.

netadm@alisa~ # ip route get 193.233.7.98

193.233.7.98 dev eth0 src 193.233.7.90

cache <redirected> mtu 1500 rtt 3072

ip rule --- routing policy database management.

Abbreviations: rule, ru.

Rules in routing policy database controlling route selection algorithm.

Classic routing algorithms used in the Internet make routing decisions based only on the destination address of packets and in theory, but not in practice, on the TOS field. In some circumstances we want to route packets differently depending not only on the destination addresses, but also on other packet fields such as source address, IP protocol, transport protocol ports or even packet payload. This task is called "policy routing".

"policy routing" != "routing policy"

"policy routing" = "cunning routing"

"routing policy" = "routing tactics" or "routing plan"

To solve this task the conventional destination based routing table, ordered according to the longest match rule, is replaced with the "routing policy database" or RPDB, which selects the appropriate route through execution of some set of rules. These rules may have many keys of different natures and therefore they have no natural ordering excepting that which is imposed by the network administrator. In Linux the RPDB is a linear list of rules ordered by a numeric priority value. The RPDB explicitly allows matching packet source address, packet destination address, TOS, incoming interface (which is packet metadata, rather than a packet field), and using fwmark values for matching IP protocols and transport ports.

Each routing policy rule consists of a selector and an action predicate. The RPDB is scanned in the order of increasing priority with the selector of each rule applied to the source address, destination address, incoming interface, tos, and fwmark. If the selector matches the packet the action is performed. The action predicate may return success in which case the rule output provides either a route or a failure indication and RPDB lookup is then terminated. Otherwise, the RPDB program continues on to the next rule.

What is the action semantically? The natural action is to select the nexthop and output device. This is the way a packet path route is selected by Cisco IOS, let us call it "match & set". In Linux the approach is more flexible as the action includes lookups in destination-based routing tables and selecting a route from these tables according to classic longest match algorithm. The "match & set" approach then becomes the simplest case of Linux route selection realized when the second level routing table contains a single default route. Remember that Linux supports multiple routing tables managed with ip route command.

At startup the kernel configures a default RPDB consisting of three rules:

1. Priority 0: Selector = match anything

Action = lookup routing table local (ID 255).

The table local is the special routing table containing high priority control routes for local and broadcast addresses.

Rule 0 is special, it cannot be deleted or overridden.

2. Priority 32766: Selector = match anything

Action = lookup routing table main (ID 254)

The table main is the normal routing table containing all non-policy routes. This rule may be deleted or overridden with other rules.

3. Priority 32767: Selector = match anything

Action = lookup routing table default (ID 253).

The table default is empty and reserved for post-processing if previous default rules did not select the packet. This rule also may be deleted.

Do not mix routing tables and rules. Rules point to routing tables, several rules may refer to one routing table and some routing tables may have no rules pointing to them. If you delete all the rules referring to a table then the table is not used but still exists. A routing table will disappear only after all the routes contained within it are deleted.

Rule attributes: Each RPDB entry has additional attributes attached. Each rule has a pointer to some routing table. NAT and masquerading rules have the attribute to select a new IP address to translate/masquearade. Additionally rules have some of the optional attributes which routes have such as realms. These values do not override those contained in routing tables, they are used only if the route did not select any of those attributes.

Rule types: The RPDB may contain rules of the following types.

unicast --- the rule prescribes returning the route found in the routing table referenced by the rule.

blackhole --- the rule prescribes to drop packet silently.

unreachable --- the rule prescribes generating the error "Network is unreachable".

prohibit --- the rule prescribes generating the error "Communication is administratively prohibited".

nat --- the rule prescribes translating the source address of the IP packet to some other value.

ip rule add --- insert new rule

Abbreviations: add, a; delete, del, d.

Arguments:

type TYPE (default) --- type of this rule. The list of valid types was given in the previous subsection.

from PREFIX --- select source prefix to match.

to PREFIX --- select destination prefix to match.

iif NAME --- select incoming device to match. If the interface is loopback, the rule matches only packets originated by this host. It means that you may create separate routing tables for forwarded and local packets and, hence, completely segregate them.

tos TOS or dsfield TOS --- select TOS value to match.

fwmark MARK --- select value of fwmark to match.

priority PREFERENCE --- priority of this rule. Each rule should have an explicitly set unique priority value. Priority is an unsigned 32 bit number thus we have 4294967296 possible rules.

WARNING!

For historical reasons ip rule add does not require any priority value and allows the priority value to be non-unique. If the user had not supplied a priority value then one was assigned by the kernel.If the user requested creating a rule with a priority value which already existed then the kernel did not reject the request and added the new rule before all old rules of the same priority. This is a mistake in the current design, nothing more. It should be fixed by the time you read this so please do not rely on this feature. You should always use explicit priorities when creating rules.

table TABLEID --- routing table identifier to lookup if the rule selector matches.

realms FROM/TO --- Realms to select if the rule matched and routing table lookup succeeded. Realm TO is used only if the route returned did not select any realm.

nat ADDRESS --- The base of IP address block to translate source address. The ADDRESS may be either the start of a block of NAT addresses as selected by NAT routes, a local host address, or even zero. In the last two cases the Linux router does not NAT translate the packets but masquerades them to this address.

Changes to the RPDB made with these commands do not become active immediately. You should run ip route flush cache to flush out the routing cache after inserting rules.

Examples:

Route packets with source addresses from 192.203.80/24 according to routing table inr.ruhep

ip rule add from 192.203.80.0/24 table inr.ruhep prio 220

Translate packet source 193.233.7.83 to 192.203.80.144 and route it according to table #1 (Table #1 is defined in /etc/iproute/rt_tables as inr.ruhep)

ip rule add from 193.233.7.83 nat 192.203.80.144 table 1 prio 320

Delete unused default rule

ip rule del prio 32767

ip rule show - list policy rules

Abbreviations: show, list, sh, ls, l.

Good news - this is the only command which has no arguments. Here is the example:

kuznet@amber~ $ ip rule list

0 from all lookup local

200 from 192.203.80.0/24 to 193.233.7.0/24 lookup main

210 from 192.203.80.0/24 to 192.203.80.0/24 lookup main

220 from 192.203.80.0/24 lookup inr.ruhep realms inr.ruhep/radio-msu

300 from 193.233.7.83 to 193.233.7.0/24 lookup main

310 from 193.233.7.83 to 192.203.80.0/24 lookup main

320 from 193.233.7.83 lookup inr.ruhep map-to 192.203.80.144

32766 from all lookup main

In the first position the rule priority value stands followed by a colon. Then the selectors follow with each key prefixed by the keyword used to create the rule.

The keyword lookup is followed by the routing table identifier as recorded in the file /etc/iproute2/rt_tables.

If the rule does NAT, as in rule #320, it is shown by the keyword map-to followed by the start of the block of addresses to map.

The sense of this example is pretty simple. The prefixes 192.203.80.0/24 and 193.233.7.0/24 form an internal network but each prefix is routed differently. Additionally, the host 193.233.7.83 is translated to another prefix as 192.203.80.144 when talking to the outer world.

ip tunnel - ip tunnelling configuration

Abbreviations: tunnel, tunl.

The tunnel objects are tunnels encapsulating packets within IPv4 packets and sending them over the IP infrastructure.

ip tunnel add - creating tunnels

Abbreviations: add, a

Arguments:

name NAME (default) --- select tunnel device name.

mode MODE --- set tunnel mode. Three modes are available: ipip, sit, gre

remote ADDRESS --- set remote endpoint of the tunnel.

local ADDRESS --- set fixed local address for tunneled packets. It must be an address on another interface of this host.

ttl N --- set fixed TTL N on tunneled packets. N is number in the range 1--255. 0 is special value, meaning that packets inherit TTL value. Default value is inherit.

tos TOS or dsfield TOS --- set fixed TOS on tunneled packets. Default value is inherit.

dev NAME --- bind tunnel to device NAME, so that tunneled packets will be routed only via this device and will not able to escape to another device, when route to endpoint changes.

nopmtudisc --- disable Path MTU Discovery on this tunnel. It is enabled by default. Note that a fixed ttl is incompatible with this option. A tunnel with fixed ttl always performs pmtu discovery.

key K, ikey K, okey K --- (GRE only) use keyed GRE with key K. K is either number or IP address-like dotted quad. The parameter key sets key to use in both directions, ikey and okey allow setting different keys for input and output.

csum, icsum, ocsum --- (GRE only) checksum tunneled packets. The flag ocsum orders checksumming outgoing packets, icsum requires that all the input packets have a correct checksum. csum is equivalent to the combination "icsum ocsum".

seq, iseq, oseq --- (GRE only) serialize packets. The flag oseq enables sequencing outgouing packets, iseq requires that all the input packets were serialized. seq is equivalent to the combination "iseq oseq".

I think this option does not work. At least, I did not test it, did not debug it and even do not understand, how it is supposed to work and for what purpose Cisco planned to use it. Do not use it. -- Alexey

Examples:

Create pointopoint IPv6 tunnel with maximal TTL of 32.

ip tunl add Cisco mode sit remote 192.31.7.104 local 192.203.80.142 ttl 32

ip tunnel show - list tunnel attributes

Abbreviations: show, list, sh, ls, l.

Example:

kuznet@amber~ $ ip tunl ls Cisco

Cisco: ipv6/ip remote 192.31.7.104 local 192.203.80.142 ttl 32

The line starts with the tunnel device name terminated by a colon then the tunnel mode follows. The parameters of the tunnel are listed with the same keywords which were used at tunnel creation.

kuznet@amber~ $ ip -s tunl ls Cisco

Cisco ipv6/ip remote 192.31.7.104 local 192.203.80.142 ttl 32

RX Packets Bytes Errors CsumErrs OutOfSeq Mcasts

12566 1707516 0 0 0 0

TX Packets Bytes Errors DeadLoop NoRoute NoBufs

13445 1879677 0 0 0 0

Essentially these numbers are the same as those printed using ip -s link show but the tags are different to reflect tunnel specific features. These features are:

CsumErrs --- total number of packets dropped because of checksum failures for GRE tunnel with enabled checksumming.

OutOfSeq --- total number of packets dropped because they arrived out of sequence for GRE tunnel with enabled serialization.

Mcasts --- total number of multicast packets, received on broadcast GRE tunnel.

DeadLoop --- total number of packets, which were not transmitted because tunnel is looped back to itself.

NoRoute --- total number of packets, which were not transmitted because there is no IP route to remote endpoint.

NoBufs --- total number of packets, which were not transmitted because kernel failed to allocate buffer.

TC Command Set

Usage: tc [OPTIONS] OBJECT { COMMAND | help }

where

```
OBJECT := { qdisc | class | filter }
OPTIONS := { -s[tatistics] | -d[etails] | -r[aw] }
```

tc qdisc

Arguments

[add | del | replace | change | get]

Usage :

Usage: tc qdisc [add | del | replace | change | get] dev STRING

[handle QHANDLE] [root | ingress | parent CLASSID]

[estimator INTERVAL TIME_CONSTANT]

[[QDISC_KIND] [help | OPTIONS]]

tc qdisc show [dev STRING] [ingress]

Where:

QDISC_KIND := { [p|b]fifo | tbf | prio | cbq | red | etc. }

OPTIONS := ... try tc qdisc add <desired QDISC_KIND> help

qdisc tbf

Usage: ... tbf limit BYTES burst BYTES[/BYTES] rate KBPS [mtu BYTES[/BYTES]] [peakrate KBPS] [latency TIME]

Description.

A data flow obeys TBF with rate R and depth B, if for any time interval t_i...t_f the number of transmitted bits does not exceed B + R*(t_f-t_i).
Packetized version of this definition:
The sequence of packets of sizes s_i served at moments t_i obeys TBF, if for any i<=k:

s_i+....+s_k <= B + R*(t_k - t_i)

qdisc cbq

Usage: ... cbq bandwidth BPS avpkt BYTES [mpu BYTES] [cell BYTES] [ewma LOG]

Where
Bandwidth : Total bandwidth of the interface

Avpkt : average length of packets

Mpu : minimum packet size

Cbq means class based queuing

Class

Usage: tc class [add | del | change | get] dev STRING

[classid CLASSID] [root | parent CLASSID]

[[QDISC_KIND] [help | OPTIONS]]

tc class show [dev STRING] [root | parent CLASSID]

Where:

QDISC_KIND := { prio | cbq | etc. }

OPTIONS := ... try tc class add <desired QDISC_KIND> help

class cbq

Usage: ... cbq bandwidth BPS rate BPS maxburst PKTS [avpkt BYTES]

[minburst PKTS] [bounded] [isolated]

[allot BYTES] [mpu BYTES] [weight RATE]

[prio NUMBER] [cell BYTES] [ewma LOG]

[estimator INTERVAL TIME_CONSTANT]

[split CLASSID] [defmap MASK/CHANGE]

Filter

Usage: tc filter [add | del | change | get] dev STRING

[pref PRIO] [protocol PROTO]

[estimator INTERVAL TIME_CONSTANT]

[root | classid CLASSID] [handle FILTERID]

[[FILTER_TYPE] [help | OPTIONS]]

tc filter show [dev STRING] [root | parent CLASSID]

Where:

FILTER_TYPE := { rsvp | u32 | fw | route | etc. }

FILTERID := ... format depends on classifier, see there

OPTIONS := ... try tc filter add <desired FILTER_KIND> help

filter u32

Usage: ... u32 [match SELECTOR ...] [link HTID] [classid CLASSID]

[police POLICE_SPEC] [offset OFFSET_SPEC]

[ht HTID] [hashkey HASHKEY_SPEC]

[sample SAMPLE]

or u32 divisor DIVISOR

Where: SELECTOR := SAMPLE SAMPLE ...

SAMPLE := { ip | ip6 | udp | tcp | icmp | u{32|16|8} } SAMPLE_ARGS

FILTERID := X:Y:Z

filter route

Usage: ... route [from REALM | fromif TAG] [to REALM]

[flowid CLASSID] [police POLICE_SPEC]

POLICE_SPEC := ... look at TBF

CLASSID := X:Y\n

Code which we have used for traffic control

```
$ tc qdisc add dev eth1 root handle 1: cbq bandwidth \
100Mbit avpkt 1000 mpu 64
$ tc class add dev eth1 parent 1:0 classid 1:2 cbq \
bandwidth 100Mbit rate 5Kbit \
avpkt 1000 bounded isolated mpu 64
$ tc filter add dev eth1 parent 1:0 protocol ip u32 \
match src ip 192.168.1.223 flowid 1:2
$ tc filter add dev eth1 parent 1:0 protocol ip route \
to 20 flowid 1:2
```

This code first defines a Queue Discipline for device eth1 ,we have choosen queue disciple cbq i.e class based queuing which have 100Mbit bandwidth , with average packet length 1000 bytes and minimum packet size 64 bytes

Then a class is made under queue disciple 1:0 which has classid 1:2 .It also have cbq with rate as 5Kbit i.e traffic under this class will get only 5Kbit bandwidth ,since it is bounded and isolated so it cannot borrow or donate it's bandwidth from parent and to children respectively .

Now we have added two filters under the class of 1:0 .

First filter uses u32 filter to check for packets which have came from 192.168.1.223 and then direct them to class 1:2 which has bandwidth 5Kbit. So packets coming from 192.168.1.223 will experience bandwidth as 5Kbit

Second filter also does same job but it select packets according to their realm .Packets which have realm set as 20 will be directed to 5Kbit bandwidth class i.e 1:2

Now to set realm of packets as 20 we you can use this code. Here this code sets the realm of packets as 20 if they have their fwmark set as 1.

```
$ ip rule add fwmark 1 table 25

$ ip route add default dev eth0 realm 20 table 25
```

Code given below set fwmark as 1 for the packets which have their destination address as 192.168.1.154

```
$ iptables -t mangle -A PREROUTING -p TCP -d
192.168.1.154 -j MARK \
--set-mark 1
```

References

Firewall (General)

1) Building Internet Firewalls
By: Zwicky, Elizabeth D.; Coooper, Simon; Chapman, D. Brent
ISBN 81-7366-101-4

2) Firewalls In 24Seven
By: Strebe, Matthew; Perkins, Charles

3) Linux Firewalls
By: Ziegler, Robert L.

4) Firewall FAQ - http://www.interhack.net/pubs/fwfaq/

5) Firewall HOWTO - http://sunsite.unc.edu/LDP/HOWTO/Firewall-HOWTO.html

6) Our own Mailing list firewall-dev@coollist.com.

IPTables Related

1) Netfilter-Hacking HOWTO (http://netfilter.samba.org/documentation/HOWTO//netfilter-hacking-HOWTO.html)

2) Packet-Filtering HOWTO (http://netfilter.samba.org/documentation/HOWTO//packet-filtering-HOWTO.html)

3) NAT-HOWTO (http://netfilter.samba.org/documentation/HOWTO//NAT-HOWTO.html)

4) Netfilter/IPTables FAQ (http://netfilter.samba.org/documentation/FAQ/netfilter-faq.html)

5) IPTables Tutorial by Oscar Andreasson (http://people.unix-fu.org/andreasson/iptables-tutorial/iptables-tutorial.html)

Advanced Routing Related

1) Advanced Routing HOWTO (http://www.tldp.org/HOWTO/Adv-Routing-HOWTO.html)

2) ipRoute2 Notes (http://snafu.freedom.org/linux2.2/iproute-notes.html)

3) ipRoute2 Documentation (http://www.linuxgrill.com/iproute2.doc.html)

Appendices

Appendix A

Sample Firewall Security Policy Configuration File

The current format definition is in version 0.3. The Changelog follows this sample file.

```
#configuration file for firewall configuration using IPTABLES
#format v 0.03
#changed by javed on 6:51 AM 4/18/02
#created by javed on 3:28 PM 3/16/2002

#format for the file
# '#' is used to signify a one-line comment
#
# action=append_rule; delete_rule; replace_rule; insert_rule;
flush_chain; zero_counters; create_chain; delete_chain;
rename_chain; policy_set
#NOTE: not allowing LIST (-L) in action
# target=ACCEPT; DROP; REJECT
# table=filter; nat; mangle
# chain=INPUT; FORWARD; OUTPUT
# protocol=ANY; constants difined in /etc/protocols
# s_interface=ANY; eth0; ppp0
# s_addr=ANY; 192.168.1.4; 192.168.1.4/255.255.255.0;
192.168.1.5/24
# d_interface=ANY; eth0; ppp0
# d_addr=ANY; 192.168.1.4; 192.168.1.4/255.255.255.0;
192.168.1.5/24
# [protocol specific information]
#
# module=state; multiport
# state=ESTABLISHED, RELATED, INVALID, NEW
# mac_source=XX:XX:XX:XX:XX:XX # This is an Ethernet Address
# limit_rate=3/hour; 2/minute; 5/second; 1/day
# limit_burst=<number>; 5; 7
# port=80,443,8080
# mark=value[/mask] # mask is logically ANDed with value
# uid_owner=userid
# gid_owner=groupid
# pid_owner=processid
# sid_owner=sessionid
# tos=16; 8; 4; 2; 0      #16=Minimize Delay
                    #8 =Maximize Throughput
                    #4 =Maximize Reliability

                    #2 =Minimize Cost
                    #0 =Normal-Service
#
#
#
# log_level=<level>
# log_prefix=<prefix>
```

```
# log_tcp_sequence=<y/n>
# log_tcp_options=<y/n>
# log_ip_options=<y/n>
#
# set_mark=<mark>
#
# reject_with=<type>
#
# set_tos=<tos>
# set_ttl=<ttl>
# ttl_decrease=<decrement>
# ttl_increase=<increment>
# ttl=<ttl>
#

# action=append_rule
# target=ACCEPT; DROP; REJECT
# table=filter; nat; mangle
# chain=INPUT; FORWARD; OUTPUT
# protocol=TCP
# s_interface=ANY; eth0; ppp0
# s_addr=192.168.1.4; 192.168.1.4/255.255.255.0; 192.168.1.5/24
# d_interface=ANY; eth0; ppp0
# d_addr=ANY; 192.168.1.4; 192.168.1.4/255.255.255.0;
192.168.1.5/24
# s_port=ANY; 20; 20:40 ; 30: ; :80
# d_port=ANY; 20; 20:40 ; 30: ; :80
# tcp_flags=ANY
# tcp_option=ANY
#

# action=append_rule
# target=ACCEPT; DROP; REJECT
# table=filter; nat; mangle
# chain=INPUT; FORWARD; OUTPUT
# protocol=UDP
# s_interface=ANY; eth0; ppp0
# s_addr=ANY; 192.168.1.4; 192.168.1.4/255.255.255.0;
192.168.1.5/24
# d_interface=ANY; eth0; ppp0
# d_addr=ANY; 192.168.1.4; 192.168.1.4/255.255.255.0;
192.168.1.5/24
# s_port=ANY; 20; 20:40 ; 30: ; :80
# d_port=ANY; 20; 20:40 ; 30: ; :80
#

# action=append_rule
# target=ACCEPT; DROP; REJECT
# table=filter; nat; mangle
# chain=INPUT; FORWARD; OUTPUT
# protocol=ICMP
# s_interface=ANY; eth0; ppp0
# s_addr=ANY; 192.168.1.4; 192.168.1.4/255.255.255.0;
192.168.1.5/24
# d_interface=ANY; eth0; ppp0
# d_addr=ANY; 192.168.1.4; 192.168.1.4/255.255.255.0;
192.168.1.5/24
```

```
# icmp_type=ANY
#

#declaration of variables

$lan_ip= 192.168.1.1

#creating a new chain
#action=create_chain
#table=filter
#chain=tcp_syn_flood


#giving access of the internal webserver to the outside world
action=append_rule
target=ACCEPT
table=filter
chain=INPUT
protocol=TCP
s_interface=ANY
s_addr=ANY
d_interface=ANY
d_addr=$lan_ip
s_port=ANY
d_port=80
tcp_flags=ANY
tcp_option=ANY

#giving access of the mail server to the outside world
action=append_rule
target=ACCEPT
table=filter
chain=INPUT
protocol=TCP
s_interface=ANY
s_addr=ANY
d_interface=ANY
d_addr=192.168.1.1
s_port=ANY
d_port=25
tcp_flags=ANY
tcp_option=ANY

#giving internal world all the access to the outside world
action=append_rule
target=ACCEPT
table=filter
chain=FORWARD
protocol=TCP
s_interface=eth1
s_addr=ANY
d_interface=ANY
d_addr=ANY
s_port=ANY
d_port=ANY
tcp_flags=ANY
tcp_option=ANY
```

```
#blocking SYN packets from the outside world
action=append_rule
target=DROP
table=filter
chain=OUTPUT
protocol=TCP
s_interface=ANY
s_addr=192.168.1.1
d_interface=ANY
d_addr=ANY
s_port=ANY
d_port=ANY
tcp_flags=SYN SYN
tcp_option=ANY
```

Changelog

```
v0.01
Initial Release

v0.02
added lots of new tags

v0.03
renamed the tag action to target
added tag action to specify a command
```

Appendix B

Source Code of fire_conf_read.cpp

The current stable version of this program is v1.6. The Changelog is given after the source code.

```
// v 1.6
// fire_conf_read.cpp : Defines the entry point for the
console application.
//

#include<stdio.h>
#include<stdlib.h>
#include<string.h>

#define conf_file "firewall.conf"

#define MAX_RULES 255
#define SIZE_RULE 1024
#define MAX_BUF 255

int readline(FILE *input,char *buffer,int len);
int readline_wout_comment(FILE *input,char *buffer,int
len,char comment_char);
int decipher_from_conf(char *buf,char
rules[MAX_RULES][SIZE_RULE],int *cur_rule,int *start_rule);
int break_line(char *buf,char *property,char *value,char
break_char);
void strip_initial_final_spaces(char *input, char *output);

class Cvariables_t
{
private:
  struct variables_t
  {
    char name[255];
    char value[255];
    variables_t *next;
  };
  variables_t *queue_start, *queue_end;
  int isadded(char *name);

public:
  Cvariables_t();
  int addvariable(char *property, char *value);
  int insert_value(char *input_buffer, char
*output_buffer);
  int getvalue(char *name, char *value);
}variables;
```

```
Cvariables_t::Cvariables_t()
{
  queue_start=queue_end=NULL;
}

// returns 0 on success
int Cvariables_t::addvariable(char *property, char *value)
{
  if(!isadded(property))
    return 1; //variable already declared
  variables_t *temp=new variables_t;
  strcpy(temp->name,property);
  strcpy(temp->value,value);

  if(queue_start==NULL) //queue empty
    {
      queue_start=queue_end=temp;
      temp->next=NULL;
    }
  else
    {
      queue_end->next=temp;
      temp->next=NULL;
      queue_end=temp;
    }
  return 0;
}

int Cvariables_t::isadded(char *name)
{
  variables_t *temp;
  temp=queue_start;
  while(temp!=NULL)
    {
      if(strcmp(name,temp->name)==0)
     return 0;//success i.e variable already present
      temp=temp->next;
    }
  return 1;//failure
}

int Cvariables_t::getvalue(char *variable,char *value)
{
  variables_t *temp;
  temp=queue_start;
  while(temp!=NULL)
    {
      if(strcmp(variable,temp->name)==0)
     {
       strcpy(value,temp->value);

       return 0;//success i.e variable already present
     }
     temp=temp->next;
```

```
    }
  return 1;//failure
}
// all buffers should be set to 0 initially using memset
int Cvariables_t::insert_value(char *input_buffer, char
*output_buffer)
{
  int i=0;char variable_name[255], value[255];
  int var_index=0,outbuffer_index=0;
  int result;
  memset(variable_name,0,255);
  memset(value,0,255);
  while(input_buffer[i]!='\0')
    {
      if(input_buffer[i]=='$')
     {
       variable_name[var_index]='$';
       i++;

while((input_buffer[i]>='A'&&input_buffer[i]<='Z')||(input_
buffer[i]>='a'&&input_buffer[i]<='z')||input_buffer[i]=='_'
)
         {
           variable_name[++var_index]=input_buffer[i++];
         }

       result=getvalue(variable_name,value);
       if(result!=0)
         return 1;
       else
         {
           strcat(output_buffer, value);
           outbuffer_index+=strlen(value);

output_buffer[outbuffer_index++]=input_buffer[i++];
         }
     }
      else
     {
       output_buffer[outbuffer_index++]=input_buffer[i++];
     }
    }
  return 0;
}

int main(int argc, char* argv[])
{
     FILE *input;
     char buf[MAX_BUF];
     char rules[MAX_RULES][SIZE_RULE];
     int start_rule,cur_rule;
     int len_buf=MAX_BUF;
     int check_deciphering=0;
```

```
     start_rule=0,cur_rule=-1;
     memset(rules,0,MAX_RULES*SIZE_RULE);

     input=fopen(conf_file,"r");
     if(input==NULL)
     {
          perror("fopen:");
          exit(1);
     }
     while(!readline_wout_comment(input,buf,len_buf,'#')&&c
heck_deciphering==0)
     {
          if(strcmp(buf,"")==0)
               printf("\n EMPTY");
          //printf("%s\n",buf);

     check_deciphering=decipher_from_conf(buf,rules,&cur_ru
le,&start_rule);
     }
     fclose(input);
     //displaying the rules

     char buffer[32767];
     memset(buffer,0,32767);
     for(int i=0;i<=cur_rule;i++)
       {
         strcat(buffer,"iptables ");
         strcat(buffer,rules[i]);
          printf("\n %s",rules[i]);
          system(buffer);
          memset(buffer,0,32767);
       }
     //getch();
     return 0;
}

/* returns 0 on success
   returns 1 on failure */
int readline(FILE *input,char *buffer,int len)
{
     char c;
     int index=0;
     while(!feof(input)&&(c=fgetc(input))!='\n'&&index<len)
     {
          buffer[index]=c;
          index++;
     }
     buffer[index]='\0';
     if(feof(input))
          return 1;
     else
          return 0;
}
```

```
/* returns 0 on success
   returns 1 on failure */
int readline_wout_comment(FILE *input,char *buffer,int
len,char comment_char)
{
     char c;
     int index=0,flag=0;

     while(!feof(input))
     {
          if((c=fgetc(input))!='\n'&&index<len)
          {
               if(c==comment_char)
               {

     while(!feof(input)&&(c=fgetc(input))!='\n');
                    if(feof(input))
                    {
                         buffer[index]='\0';
                         return 1;
                    }
                    index=0;
                    flag=1;
               }
               if(flag==0)
               {
                    buffer[index]=c;
                    index++;
               }
               if(flag==1)
                    flag=0;
          }
          else
               if(c=='\n'&&index>0)
                    break;
     }
     buffer[index]='\0';
     if(feof(input))
          return 1;
     else
          return 0;
}

/* Build the rules from the conf file */
int decipher_from_conf(char *buf,char
rules[MAX_RULES][SIZE_RULE],int *cur_rule,int *start_rule)
{

        static int action_position=0;
        char
property[MAX_BUF],value[MAX_BUF],temp_buffer[MAX_BUF];
     int check_break,i,j,rule_len;
     memset(property,0,MAX_BUF);
     memset(value,0,MAX_BUF);
```

```
memset(temp_buffer,0,MAX_BUF);
check_break=break_line(buf,property,value,'=');

//strcpy(property,strlwr(property));

if(check_break!=0)
     return 1;
strip_initial_final_spaces(property,temp_buffer);
strcpy(property,temp_buffer);
memset(temp_buffer,0,MAX_BUF);
strip_initial_final_spaces(value,temp_buffer);
strcpy(value,temp_buffer);

if(property[0]=='$')
  {
    int result;
    result=variables.addvariable(property,value);
    if(result!=0)
      {
     printf("Error in declaration of %s", property);
     exit(1);  //change this to return 1 in future
               //versions
      }
    return 0;
  }
memset(temp_buffer,0,MAX_BUF);
variables.insert_value(value,temp_buffer);
strcpy(value,temp_buffer);

if(strcmp(property,"action")==0)
{
  (*cur_rule)++;
  if(strcmp(value,"append_rule")==0)
    {
      strcat(rules[*cur_rule]," -A ");
      action_position=strlen(rules[*cur_rule]);
      return 0;
    }
  if(strcmp(value,"delete_rule")==0)
    {
      strcat(rules[*cur_rule]," -D ");
      action_position=strlen(rules[*cur_rule]);
      return 0;
    }
  if(strcmp(value,"replace_rule")==0)
    {
      strcat(rules[*cur_rule]," -R ");
      action_position=strlen(rules[*cur_rule]);
      return 0;
    }
  if(strcmp(value,"insert_rule")==0)
    {
      strcat(rules[*cur_rule]," -I ");
      action_position=strlen(rules[*cur_rule]);
```

```
      return 0;
    }
  if(strcmp(value,"flush_chain")==0)
    {
      strcat(rules[*cur_rule]," -F ");
      action_position=strlen(rules[*cur_rule]);
      return 0;
    }
  if(strcmp(value,"create_chain")==0)
    {
      strcat(rules[*cur_rule]," -N ");
      action_position=strlen(rules[*cur_rule]);
      return 0;
    }
  if(strcmp(value,"delete_chain")==0)
    {
      strcat(rules[*cur_rule]," -X ");
      action_position=strlen(rules[*cur_rule]);
      return 0;
    }
  if(strcmp(value,"policy_set")==0)
    {
      strcat(rules[*cur_rule]," -P ");
      action_position=strlen(rules[*cur_rule]);
      return 0;
    }
  if(strcmp(value,"zero_counters")==0)
    {
      strcat(rules[*cur_rule]," -Z ");
      action_position=strlen(rules[*cur_rule]);
      return 0;
    }
  return 1;
}
if(strcmp(property,"target")==0)
{
  strcat(rules[*cur_rule]," -j ");
  strcat(rules[*cur_rule],value);
  return 0;
}
if(strcmp(property,"chain")==0)
{
  //strcat(rules[*cur_rule]," -A ");
  memset(temp_buffer,0,MAX_BUF);
  for(i=0;i<action_position; i++)
    temp_buffer[i]=rules[*cur_rule][i];
  strcat(temp_buffer,value);
  j=i+strlen(value);
  rule_len=strlen(rules[*cur_rule]);
  for(; i<rule_len;i++,j++)
  {
    temp_buffer[j]=rules[*cur_rule][i];
  }
  strcpy(rules[*cur_rule], temp_buffer);
```

```
  memset(temp_buffer,0,MAX_BUF);
  return 0;
}
if(strcmp(property,"protocol")==0)
{
  if(strcmp(value,"ANY")!=0)
    {
      strcat(rules[*cur_rule]," -p ");
      strcat(rules[*cur_rule],value);
    }
  return 0;
}
if(strcmp(property,"s_interface")==0)
{
  if(strcmp(value,"ANY")!=0)
  {
    strcat(rules[*cur_rule]," -i ");
    strcat(rules[*cur_rule],value);
  }
  return 0;
}
if(strcmp(property,"d_interface")==0)
{
  if(strcmp(value,"ANY")!=0)
  {
    strcat(rules[*cur_rule]," -o ");
    strcat(rules[*cur_rule],value);
  }
  return 0;
   }
if(strcmp(property,"s_addr")==0)
{
  if(strcmp(value,"ANY")!=0)
  {
    strcat(rules[*cur_rule]," -s ");
    strcat(rules[*cur_rule],value);
  }
  return 0;
   }
if(strcmp(property,"d_addr")==0)
{
  if(strcmp(value,"ANY")!=0)
  {
    strcat(rules[*cur_rule]," -d ");
    strcat(rules[*cur_rule],value);
  }
  return 0;
   }
if(strcmp(property,"s_port")==0)
{
  if(strcmp(value,"ANY")!=0)
  {
    strcat(rules[*cur_rule]," --source-port ");
    strcat(rules[*cur_rule],value);
```

```
  }
  return 0;
}
if(strcmp(property,"d_port")==0)
{
  if(strcmp(value,"ANY")!=0)
  {
    strcat(rules[*cur_rule]," --destination-port ");
    strcat(rules[*cur_rule],value);
  }
  return 0;
}
if(strcmp(property,"tcp_flags")==0)
{
  if(strcmp(value,"ANY")!=0)
  {
    strcat(rules[*cur_rule]," --tcp-flags ");
    strcat(rules[*cur_rule],value);
  }
  return 0;
}
if(strcmp(property,"table")==0)
{
  if(strcmp(value,"ANY")!=0)
  {
    strcat(rules[*cur_rule]," -t ");
    strcat(rules[*cur_rule],value);
  }
  return 0;
    }
if(strcmp(property,"tcp_option")==0)
{
  if(strcmp(value,"ANY")!=0)
  {
    strcat(rules[*cur_rule]," -tcp-option ");
    strcat(rules[*cur_rule],value);
  }
  return 0;
    }
if(strcmp(property,"icmp_type")==0)
{
  if(strcmp(value,"ANY")!=0)
  {
    strcat(rules[*cur_rule]," -icmp-type ");
    strcat(rules[*cur_rule],value);
  }
  return 0;
    }
if(strcmp(property,"module")==0)
{
  if(strcmp(value,"ANY")==0)
    {
      return 1;
    }
```

```
  strcat(rules[*cur_rule]," -m ");
  strcat(rules[*cur_rule],value);
  return 0;
}
if(strcmp(property,"state")==0)
{
  if(strcmp(value,"ANY")!=0)
    {
      strcat(rules[*cur_rule]," --state ");
      strcat(rules[*cur_rule],value);
    }
  return 0;
}
if(strcmp(property,"mac_source")==0)
{
  if(strcmp(value,"ANY")!=0)
    {
      strcat(rules[*cur_rule]," --mac-source ");
      strcat(rules[*cur_rule],value);
    }
  return 0;
}
if(strcmp(property,"limit_rate")==0)
{
  if(strcmp(value,"ANY")!=0)
    {
      strcat(rules[*cur_rule]," --limit ");
      strcat(rules[*cur_rule],value);
    }
  return 0;
}
if(strcmp(property,"limit_burst")==0)
{
  if(strcmp(value,"ANY")!=0)
    {
      strcat(rules[*cur_rule]," --limit-burst ");
      strcat(rules[*cur_rule],value);
    }
  return 0;
}
if(strcmp(property,"port")==0)
{
  if(strcmp(value,"ANY")!=0)
    {
      strcat(rules[*cur_rule]," --port ");
      strcat(rules[*cur_rule],value);
    }
  return 0;
}
if(strcmp(property,"mark")==0)
{
  if(strcmp(value,"ANY")!=0)
    {
      strcat(rules[*cur_rule]," --mark ");
```

```
      strcat(rules[*cur_rule],value);
    }
  return 0;
}
if(strcmp(property,"uid_owner")==0)
{
  if(strcmp(value,"ANY")!=0)
    {
      strcat(rules[*cur_rule]," --uid-owner ");
      strcat(rules[*cur_rule],value);
    }
  return 0;
}
if(strcmp(property,"gid_owner")==0)
{
  if(strcmp(value,"ANY")!=0)
    {
      strcat(rules[*cur_rule]," --gid-owner ");
      strcat(rules[*cur_rule],value);
    }
  return 0;
}
if(strcmp(property,"pid_owner")==0)
{
  if(strcmp(value,"ANY")!=0)
    {
      strcat(rules[*cur_rule]," --pid-owner ");
      strcat(rules[*cur_rule],value);
    }
  return 0;
}
if(strcmp(property,"sid_owner")==0)
{
  if(strcmp(value,"ANY")!=0)
    {
      strcat(rules[*cur_rule]," --sid-owner ");
      strcat(rules[*cur_rule],value);
    }
  return 0;
}
if(strcmp(property,"tos")==0)
{
  if(strcmp(value,"ANY")!=0)
    {
      strcat(rules[*cur_rule]," --tos ");
      strcat(rules[*cur_rule],value);
    }
  return 0;
}
if(strcmp(property,"log_level")==0)
{
  if(strcmp(value,"ANY")!=0)
    {
      strcat(rules[*cur_rule]," --log-level ");
```

```
      strcat(rules[*cur_rule],value);
    }
  return 0;
}
if(strcmp(property,"log_prefix")==0)
{
  if(strcmp(value,"ANY")!=0)
    {
      strcat(rules[*cur_rule]," --log-prefix ");
      strcat(rules[*cur_rule],value);
    }
  return 0;
}
if(strcmp(property,"log_tcp_sequence")==0)
{
  if(strcmp(value,"y")==0||strcmp(value,"Y")==0)
    {
      strcat(rules[*cur_rule]," --log-tcp-sequence ");
    }
  return 0;
}
if(strcmp(property,"log_tcp_options")==0)
{
  if(strcmp(value,"y")==0||strcmp(value,"Y")==0)
    {
      strcat(rules[*cur_rule]," --log-tcp-options ");
      strcat(rules[*cur_rule],value);
    }
  return 0;
}
if(strcmp(property,"log_ip_options")==0)
{
  if(strcmp(value,"y")==0||strcmp(value,"Y")==0)
    {
      strcat(rules[*cur_rule]," --log-ip-options ");
      strcat(rules[*cur_rule],value);
    }
  return 0;
}
if(strcmp(property,"set_mark")==0)
{
  strcat(rules[*cur_rule]," --set-mark ");
  strcat(rules[*cur_rule],value);
  return 0;
}
if(strcmp(property,"reject_with")==0)
{
  strcat(rules[*cur_rule]," --reject-with ");
  strcat(rules[*cur_rule],value);
  return 0;
}
if(strcmp(property,"set_tos")==0)
{
  strcat(rules[*cur_rule]," --set-tos ");
```

```
      strcat(rules[*cur_rule],value);
      return 0;
    }
    if(strcmp(property,"set_ttl")==0)
    {
      strcat(rules[*cur_rule]," --ttl-set ");
      strcat(rules[*cur_rule],value);
      return 0;
    }
    if(strcmp(property,"ttl_decrease")==0)
    {
      strcat(rules[*cur_rule]," --ttl-dec ");
      strcat(rules[*cur_rule],value);
      return 0;
    }
    if(strcmp(property,"ttl_increase")==0)
    {
      strcat(rules[*cur_rule]," --ttl-inc ");
      strcat(rules[*cur_rule],value);
      return 0;
    }
     if(strcmp(property,"ttl")==0)
    {
      if(strcmp(value,"ANY")!=0)
        {
          strcat(rules[*cur_rule]," --ttl ");
          strcat(rules[*cur_rule],value);
        }
      return 0;
    }
    return 0;
}

int break_line(char *buf,char *property,char *value,char
break_char)
{
    char *ptr=strchr(buf,break_char);
    if(ptr==NULL)
        return 1;
    else
    {
        strcpy(value,ptr+1);
        strncpy(property,buf,ptr-buf);
        return 0;
    }
}
//buffers to be memset to 0 initially
void strip_initial_final_spaces(char *input, char *output)
{
  int input_index,output_index;
  input_index=output_index=0;

  while(input[input_index]==' '||input[input_index]=='\t')
    {
```

```
      input_index++;
    }

  while(input[input_index]!='\0')
    {
      output[output_index++]=input[input_index++];
    }
  output_index--;
  while(output[output_index]==' ')
    {
      output[output_index--]='\0';
    }
}
```

Changelog

```
2002-04-19  javed shakeel  <jshakeel@hs01.b2k>

      * fire_conf_read.cpp:
      Added support to delete, replace, insert a rule. Creation,
deletion of a chain.
      Flushing a chain or a table, setting the policy of a chain and
resettign the counters of a chain. Added function
strip_initial_final_spaces to fix the bug when some blank spaces are
present at the beginning or the end.

2002-04-17  javed shakeel  <jshakeel@hs01.b2k>

      * fire_conf_read.cpp: Added support for variables/constants
      Stripping of starting and end whitespace/tab characters

2002-04-17  javed shakeel  <jshakeel@hs01.b2k>

      * fire_conf_read.cpp: Added support for variables/constants
      Stripping of starting and end whitespace/tab characters

2002-04-16  javed shakeel  <jshakeel@hs01.b2k>

      * fire_conf_read.cpp:
      added class Cvariables_t for support for variables in the conf
file

2002-04-13  javed shakeel  <jshakeel@hs01.b2k>

      * fire_conf_read.cpp: compilation error checked

      * fire_conf_read.cpp: Added the following tags:
      module, state, mac_source, limit_rate, limit_burst, port, mark,
uid_owner, gid_owner, pid_owner, sid_owner, tos, log_level, log_prefix,
log_tcp_sequence, log_tcp_options, log_ip_options, set_mark,
reject_with, set_tos, set_ttl, ttl_decrease, ttl_increase, ttl

2002-04-13  javed shakeel  <jshakeel@hs01.b2k>
```

```
      * fire_conf_read.cpp: Added the following tags:
      module, state, mac_source, limit_rate, limit_burst, port, mark,
uid_owner, gid_owner, pid_owner, sid_owner, tos, log_level, log_prefix,
log_tcp_sequence, log_tcp_options, log_ip_options, set_mark,
reject_with, set_tos, set_ttl, ttl_decrease, ttl_increase, ttl

2002-04-12  javed shakeel  <jshakeel@hs01.b2k>

      * fire_conf_read.cpp: icmp_type added

      * fire_conf_read.cpp: New file.
```