IP Filter Examples

Permission Specifying.

To specify where to pass through or to block a packet, either **block** or **pass** is used. **In** and **out** are used to describe the direction in which the packet is travelling through a network interface. Eg:

```
# setup default to block all packets.
block in all
block out all
# pass packets from host firewall to any destination
pass in from firewall to any
```

Select network Interfaces

To select which interface a packet is currently associated with, either its destination as a result of route processing or where it has been received from, the **on** keyword is used. Whilst not compulsory, it is recommended that each rule include it for clarity. Eg:

```
# drop all inbound packets from localhost coming from ethernet
block in on le0 from localhost to any
```

Netmasks and hosts

As not all networks are formed with classical network boundaries, it is necessary to provide a mechanism to support VLSM (Variable Length Subnet Masks). This package provides several ways to do this. Eg:

```
#
block in on le0 from mynet/26 to any
#
block in on le0 from mynet/255.255.255.192 to any
#
block in on le0 from mynet mask 255.255.255.192 to any
#
block in on le0 from mynet mask 0xffffffc0 to any
```

Are all valid and legal syntax with this package. However, when regenerating rules (ie using ipfstat), this package will prefer to use the shortest valid notation (top down).

The default netmask, when none is given is 255.255.255.255 or "/32".

To invert the match on a hostname or network, include an ! before the name or number with no space between them.

Protocol

To filter on an individual protocol, it is possible to specify the protocol in a filter rule. Eg:

```
# block all incoming ICMP packets
block in on le0 proto icmp all
```

The name of the protocol can be any valid name from /etc/protocols or a number.

```
# allow all IP packets in which are protocol 4
pass in on le0 proto 4 all
```

There is one exception to this rule, being "tcp/udp". If given in a ruleset, it will match either of the two protocols. This is useful when setting up port restrictions. Eg:

```
# prevent any packets destined for NFS from coming in
block in on le0 proto tcp/udp from any to any port = 2049
```

Filtering IP fragments

IP fragments are bad news, in general. Recent study has shown that IP fragments can pose a large threat to Internet firewalls, IF there are rules used which rely on data which may be distributed across fragments. To this package, the threat is that the TCP flags field of the TCP packet may be in the 2nd or 3rd fragment or possibly be believed to be in the first when actually in the 2nd or 3rd.

To filter out these nasties, it is possible to select fragmented packets out as follows:

```
#
# get rid of all IP fragments
#
block in all with frag
```

The problem arises that fragments can actually be a non-malicious. The **really malicious** ones can be grouped under the term "short fragments" and can be filtered out as follows:

```
#
# get rid of all short IP fragments (too small for valid comparison)
#
block in proto tcp all with short
```

IP Options

IP options have a bad name for being a general security threat. They can be of some use, however, to programs such as **traceroute** but many find this usefulness not worth the risk.

Filtering on IP options can be achieved two ways. The first is by naming them collectively and is done as follows:

```
#
# drop and log any IP packets with options set in them.
#
block in log all with ipopts
#
```

The second way is to actually list the names of the options you wish to filter.

```
#
# drop any source routing options
#
block in quick all with opt lsrr
block in quick all with opt ssrr
```

● **NOTE** that options are matched explicitly, so if I had **lsrr,ssrr** it would only match packets with both options set.

It is also possible to select packets which **DON'T** have various options present in the packet header. For example, to allow telnet connections without any IP options present, the following would be done:

```
#
# Allow anyone to telnet in so long as they don't use IP options.
#
pass in proto tcp from any to any port = 23 with no ipopts
#
# Allow packets with strict source routing and no loose source routing
#
pass in from any to any with opt ssrr not opt lsrr
```

Filtering by ports

Filtering by port number only works with the TCP and UDP IP protocols. When specifying port numbers, either the number or the service name from /etc/services may be used. If the **proto** field is used in a filter rule, it will be used in conjunction with the port name in determining the port number.

The possible operands available for use with port numbers are:

```
Operand Alias   Parameters      Result
<       lt      port#           true if port is less than given value
>       gt      port#           true if port is greater than given
value
=       eq      port#           true if port is equal to than given
value
!=      ne      port#           true if port is not equal to than
given value
<=      le      port#           true if port is less than or equal to
given value
=>      ge      port#           true if port is greater than or equal
to given value
```

Eg:

```
#
# allow any TCP packets from the same subnet as foo is on through to
host
# 10.1.1.2 if they are destined for port 6667.
#
pass in proto tcp from fubar/24 to 10.1.1.2/32 port = 6667
#
# allow in UDP packets which are NOT from port 53 and are destined for
# localhost
#
pass in proto udp from fubar port != 53 to localhost
```

Two range comparisons are also possible:

```
Expression Syntax:
port1#  <>      port2#          true if port is less than port1 or
greater than port2
port1#  ><      port2#          true if port is greater than port1
and less than port2
```

● **NOTE** that in neither case, when the port number is equal to one of those given, does it match. Eg:

```
#
# block anything trying to get to X terminal ports, X:0 to X:9
#
block in proto tcp from any to any port 5999 >< 6010
#
# allow any connections to be made, except to BSD print/r-services
# this will also protect syslog.
#
block in proto tcp/udp all
pass in proto tcp/udp from any to any port 512 <> 515
```

Note that the last one above could just as easily be done in the reverse fashion: allowing everything through and blocking only a small range. Note that the port numbers are different, however, due to the difference in the way they are compared.

```
#
# allow any connections to be made, except to BSD print/r-services
# this will also protect syslog.
#
pass in proto tcp/udp all
block in proto tcp/udp from any to any port 511 >< 516
```

TCP Flags (established)

Filtering on TCP flags is useful, but fraught with danger. I'd recommend that before using TCP flags in your IP filtering, you become at least a little bit acquainted with what the role of each of them is and when they're used. This package will compare the flags present in each TCP packet, if asked, and match if those present in the TCP packet are the same as in the IP filter rule.

Some IP filtering/firewall packages allow you to filter out TCP packets which belong to an "established" connection. This is, simply put, filtering on packets which have the ACK bit set. The ACK bit is only set in packets transmitted during the lifecycle of a TCP connection. It is necessary for this flag to be present from either end for data to be transferred. If you were using a rule which as worded something like:

```
allow proto tcp 10.1.0.0 255.255.0.0 port = 23 10.2.0.0 255.255.0.0
```

```
established
```

It could be rewritten as:

```
pass in proto tcp 10.1.0.0/16 port = 23 10.2.0.0/16 flags A/A
pass out proto tcp 10.1.0.0/16 port = 23 10.2.0.0/16 flags A/A
```

A more useful flag to filter on, for TCP connections, I find, is the **SYN** flag. This is only set during the initial stages of connection negotiation, and for the very first packet of a new TCP connection, it is the only flag set. At all other times, an **ACK** or maybe even an **URG**/**PUSH** flag may be set. So, if I want to stop connections being made to my internal network (10.1.0.0) from the outside network, I might do something like:

```
#
# block incoming connection requests to my internal network from the
big bad
# internet.
#
block in on le0 proto tcp from any to 10.1.0.0/16 flags S/SA
```

If you wanted to block the replies to this (the SYN-ACK's), then you might do:

```
block out on le0 proto tcp from 10.1.0.0 to any flags SA/SA
```

where SA represents the **SYN-ACK** flags both being set.

The flags after the / represent the **TCP flag mask**, indicating which bits of the TCP flags you are interested in checking. When using the **SYN** bit in a check, you **SHOULD** specify a mask to ensure that your filter CANNOT be defeated by a packet with SYN and URG flags, for example, set (to Unix, this is the same as a plain SYN).

ICMP Type/Code

ICMP can be a source of a lot of trouble for Internet Connected networks. Blocking out all ICMP packets can be useful, but it will disable some otherwise useful programs, such as "ping". Filtering on ICMP type allows for pings (for example) to work. Eg:

```
# block all ICMP packets.
#
```

```
block in proto icmp all
#
# allow in ICMP echos and echo-replies.
#
pass in on le1 proto icmp from any to any icmp-type echo
pass in on le1 proto icmp from any to any icmp-type echorep
```

To specify an ICMP code, the numeric value must be used. So, if we wanted to block all port-unreachables, we would do:

```
#
# block all ICMP destination unreachable packets which are port-
unreachables
#
block in on le1 proto icmp from any to any icmp-type unreach code 3
```

Responding to a BAD packet

To provide feedback to people trying to send packets through your filter which you wish to disallow, you can send back either an ICMP error (Destination Unreachable) or, if they're sending a TCP packet, a TCP RST (Reset).

What's the difference ? TCP/IP stacks take longer to pass the ICMP errors back, through to the application, as they can often be due to temporary problems (network was unplugged for a second) and it is `incorrect' to shut down a connection for this reason. Others go to the other extreme and will shut down all connections between the two hosts for which the ICMP error is received. The TCP RST, however, is for only *one* connection (cannot be used for more than one) and will cause the connection to immediately shut down. So, for example, if you're blocking port 113, and setup a rule to return a TCP RST rather than nothing or an ICMP packet, you won't experience any delay if the other end was attempting to make a connection to an identd service.

Some examples are as follows:

```
#
# block all incoming TCP connections but send back a TCP-RST for ones
to
# the ident port
#
block in proto tcp from any to any flags S/SA
block return-rst in quick proto tcp from any to any port = 113 flags
```

```
S/SA
#
# block all inbound UDP packets and send back an ICMP error.
#
block return-icmp in proto udp from any to any
```

When returning ICMP packets, it is also possible to specify the type of ICMP error return. This was requested so that **traceroute** traces could be forced to end elegantly. To do this, the requested ICMP Unreachable code is placed in brackets following the "return-icmp" directive:

```
#
# block all inbound UDP packets and send back an ICMP error.
#
block return-icmp (3) in proto udp from any to any port > 30000
block return-icmp (port-unr) in proto udp from any to any port > 30000
```

Those two examples are equivalent, and return a ICMP port unreachable error packet to in response to any UDP packet received destined for a port greater than 30,000.

Filtering IP Security Classes

For users who have packets which contain IP security bits, filtering on the defined classes and authority levels is supported. Currently, filtering on 16bit authority flags is not supported.

As with ipopts and other IP options, it is possible to say that the packet only matches if a certain class isn't present.

Some examples of filtering on IP security options:

```
#
# drop all packets without IP security options
#
block in all with no opt sec
#
# only allow packets in and out on le0 which are top secret
#
block out on le1 all
pass out on le1 all with opt sec-class topsecret
block in on le1 all
pass in on le1 all with opt sec-class topsecret
```

Packet state filtering

Packet state filtering can be used for any TCP flow to short-cut later filtering. The "short-cuts" are kept in a table, with no alterations to the list of firewall rules. Subsequent packets, if a matching packet is found in the table, are not passed through the list. For TCP flows, the filter will follow the ack/sequence numbers of packets and only allow packets through which fall inside the correct window.

```
#
# Keep state for all outgoing telnet connections
# and disallow all other TCP traffic.
#
pass out on le1 proto tcp from any to any port = telnet keep state
block out on le1 all
```

For UDP packets, packet exchanges are effectively stateless. However, if a packet is first sent out from a given port, a reply is usually expected in answer, in the `reverse' direction.

```
#
# allow UDP replies back from name servers
#
pass out on le1 proto udp from any to any port = domain keep state
```

Held UDP state is timed out, as is TCP state for entries added which do not have the SYN flag set. If an entry is created with the SYN flag set, any subsequent matching packet which doesn't have this flag set (ie a SYN-ACK) will cause it to be "timeless" (actually, the timeout defaults to 5 days), until either a FIN or RST is seen.

Network Address Translation (NAT)

Network address translation is used to remap IP #'s from one address range to another range of network addresses. For TCP and UDP, this also can include the port numbers. The IP#'s/port #'s are changed when a packet is going out through an interface and IP Filter matches it against a NAT rules.

Packets coming back in the same interface are remapped, as a matter of course, to their original address information.

```
# map all tcp connections from 10.1.0.0/16 to 240.1.0.1, changing the
source
# port number to something between 10,000 and 20,000 inclusive.  For
all other
# IP packets, allocate an IP # between 240.1.0.0 and 240.1.0.255,
temporarily
# for each new user.  In this example, ed1 is the external interface.
# Use ipnat, not ipf to load these rules.
#
map ed1 10.1.0.0/16 -> 240.1.0.1/32 portmap tcp 10000:20000
map ed1 10.1.0.0/16 -> 240.1.0.0/24
```

Transparent Proxy Support

Transparent proxies are supported through redirection, which works in a similar way to NAT, except that rules are triggered by input packets. To effect redirection rules, **ipnat** must be used (same as for NAT) rather than **ipf**.

```
# Redirection is triggered for input packets.
# For example, to redirect FTP connections through this box (in this
case ed0
# is the interface on the "inside" where default routes point), to
the local
# ftp port, forcing them to connect through a proxy, you would use:
#
rdr ed0 0.0.0.0/0 port ftp -> 127.0.0.1 port ftp
```

Transparent routing

Transparent routing can be performed in two ways using IP Filter. The first is to use the keyword "fastroute" in a rule, using the normal route lookup to occur or using a fixed route with "to". Both effect transparent routing by not causing any decrement in the TTL to occur as it passes through the kernel.

```
# Route all UDP packets through transparently.
#
pass in quick fastroute proto udp all
```

```
#
# Route all ICMP packets to network 10 (on le0) out through le1, to
"router"
#
pass in quick on le0 to le1:router proto icmp all
```

Logging packets to the network

Logging packets to the network devices is supported for both packets being passed through the filter and those being blocked. For packets being passed on, the "dup-to" keyword must be used, but for packets being blocked, either "to" (more efficient) or "dup-to" can be used.

To log packets to the interface without requiring ARP to work, create a static arp cache for a meaningless IP# (say 10.0.0.1) and log packets to this IP#.

```
# Log all short TCP packets to qe3, with "packetlog" as the intended
# destination for the packet.
#
block in quick to qe3:packetlog proto tcp all with short
#
# Log all connection attempts for TCP
#
pass in quick on ppp0 dup-to le1:packetlog proto tcp all flags S/SA
```

Rule groups

To aide in making rule processing more efficient, it is possible to setup rule `groups'. By default, all rules are in group 0 and all other groups have it as their ultimate parent. To start a new group, a rule includes a `head' statement, such as this:

```
# Process all incoming ppp packets on ppp0 with group 100, with the
default for
# this interface to block all incoming.
#
block in quick on ppp0 all head 100
```

If we then wanted to allow people to connect to our WWW server, via ppp0, we could then just add a rule about WWW. NOTE: only packets which match the above rule are processed by any group 100 rules.

```
# Allow connections to the WWW server via ppp0.
#
pass in quick proto tcp from any to any port = WWW keep state group
100
```

Return to the IP Filter home page