NAME

ipf, ipf.conf - IPFilter firewall rules file format

DESCRIPTION

The ipf.conf file is used to specify rules for the firewall, packet authentication and packet accounting components of IPFilter. To load rules specified in the ipf.conf file, the ipf(8) program is used.

For use as a firewall, there are two important rule types: those that block and drop packets (block rules) and those that allow packets through (pass rules.) Accompanying the decision to apply is a collection of statements that specify under what conditions the result is to be applied and how.

The simplest rules that can be used in ipf.conf are expressed like this:

```
block in all
pass out all
```

Each rule must contain at least the following three components

* a decision keyword (pass, block, etc.)
* the direction of the packet (in or out)
* address patterns or "all" to match any address information

Long lines

For rules lines that are particularly long, it is possible to split them over multiple lines implicity like this:

```
pass in on bgeo proto tcp from 1.1.1.1 port > 1000
   to 2.2.2.2 port < 5000 flags S keep state
```

or explicitly using the backslash (’\’) character:

```
pass in on bgeo proto tcp from 1.1.1.1 port > 1000 \
   to 2.2.2.2 port < 5000 flags S keep state
```

Comments

Comments in the ipf.conf file are indicated by the use of the ’#’ character. This can either be at the start of the line, like this:

```
# Allow all ICMP packets in
pass in proto icmp from any to any
```

Or at the end of a like, like this:

```
pass in proto icmp from any to any # Allow all ICMP packets in
```

Firewall rules

This section goes into detail on how to construct firewall rules that are placed in the ipf.conf file.

It is beyond the scope of this document to describe what makes a good firewall rule set or which packets should be blocked or allowed in. Some suggestions will be provided but further reading is expected to fully understand what is safe and unsafe to allow in/out.

Filter rule keywords

The first word found in any filter rule describes what the eventual outcome of a packet that matches it will be. Descriptions of the many and various sections that can be used to match on the contents of packet headers will follow on below.

The complete list of keywords, along with what they do is as follows:

pass rules that match a packet indicate to ipfilter that it should be allowed to continue on in the direction it is flowing.

block rules are used when it is desirable to prevent a packet from going any further. Packets that are blocked on the "in" side are never seen by TCP/IP and those that are blocked going "out" are never seen on the wire.

log when IPFilter successfully matches a packet against a log rule a log record is generated and made available for ipmon(8) to read. These rules have no impact on whether or not a packet is allowed through or not. So if a packet first matched a block rule and then matched a log rule, the status of the packet after the log rule is that it will still be blocked.

count rules provide the administrator with the ability to count packets and bytes that match the criteria laid out in the configuration file. The count rules are applied after NAT and filter rules on the inbound path. For outbound packets, count rules are applied before NAT and before the packet is dropped. Thus the count rule cannot be used as a true indicator of link layer

auth rules cause the matching packet to be queued up for processing by a user space program. The user space program is responsible for making an ioctl system call to collect the information

about the queued packet and another ioctl system call to return the verdict (block, pass, etc) on what to do with the packet. In the event that the queue becomes full, the packets will end up being dropped.

call provides access to functions built into IPFilter that allow for more complex actions to be taken as part of the decision making that goes with the rule.

decapsulate rules instruct ipfilter to remove any other headers (IP, UDP, AH) and then process what is inside as a new packet. For non-UDP packets, there are builtin checks that are applied in addition to whatever is specified in the rule, to only allow decapsulation of recognised protocols. After decapsulating the inner packet, any filtering result that is applied to the inner packet is also applied to the other packet.

The default way in which filter rules are applied is for the last matching rule to be used as the decision maker. So even if the first rule to match a packet is a pass, if there is a later matching rule that is a block and no further rules match the packet, then it will be blocked.

Matching Network Interfaces

On systems with more than one network interface, it is necessary to be able to specify different filter rules for each of them. In the first instance, this is because different networks will send us packets via each network interface but it is also because of the hosts, the role and the resulting security policy that we need to be able to distinguish which network interface a packet is on.

To accommodate systems where the presence of a network interface is dynamic, it is not necessary for the network interface named in a filter rule to be present in the system when the rule is loaded. This can lead to silent errors being introduced and unexpected behaviour with the simplest of keyboard mistakes - for example, typing in hem0 instead of hme0 or hme2 instead of hme3.

On Solaris systems prior to Solaris 10 Update 4, it is not possible to filter packets on the loopback interface (lo0) so filter rules that specify it will have no impact on the corresponding flow of packets. See below for Solaris specific tips on how to enable this.

Some examples of including the network interface in filter rules are:

```
block in on bge0 all
pass out on bge0 all
```

Address matching (basic)

The first and most basic part of matching for filtering rules is to specify IP addresses and TCP/UDP port numbers. The source address information is matched by the "from" information in a filter rule and

the destination address information is matched with the "to" information in a filter rule.

The typical format used for IP addresses is CIDR notation, where an IP address (or network) is followed by a ’/’ and a number representing the size of the netmask in bits. This notation is used for specifying address matching in both IPv4 and IPv6. If the ’/’ and bitmask size are excluded from the matching string, it is assumed that the address specified is a host address and that the netmask applied should be all 1’s.

Some examples of this are:

pass in from 10.1.0.0/24 to any
block out from any to 10.1.1.1

It is not possible to specify a range of addresses that does not have a boundary that can be defined by a standard subnet mask.

Names instead of addresses

Hostnames, resolved either via DNS or /etc/hosts, or network names, resolved via /etc/networks, may be used in place of actual addresses in the filter rules. WARNING: if a hostname expands to more than one address, only the *first* is used in building the filter rule.

Caution should be exercised when relying on DNS for filter rules in case the sending and receiving of DNS packets is blocked when ipf(8) is processing that part of the configuration file, leading to long delays, if not errors, in loading the filter rules.

Protocol Matching

To match packets based on TCP/UDP port information, it is first necessary to indicate which protocol the packet must be. This is done using the "proto" keyword, followed by either the protocol number or a name which is mapped to the protocol number, usually through the /etc/protocols file.

pass in proto tcp from 10.1.0.0/24 to any
block out proto udp from any to 10.1.1.1
pass in proto icmp from any to 192.168.0.0/16

Sending back error packets

When a packet is just discarded using a block rule, there is no feedback given to the host that sent the packet. This is both good and bad. If this is the desired behaviour and it is not desirable to send any feedback about packets that are to be denied. The catch is that often a host trying to connect to a TCP port or with a UDP based application will send more than one packet because it assumes that just one

packet may be discarded so a retry is required. The end result being logs can become cluttered with duplicate entries due to the retries.

To address this problem, a block rule can be qualified in two ways. The first of these is specific to TCP and instructs IPFilter to send back a reset (RST) packet. This packet indicates to the remote system that the packet it sent has been rejected and that it shouldn't make any further attempts(to send packets to that port. Telling IPFilter to return a TCP); RST packet in response to something that has been received is achieved with the return-rst keyword like this:

block return-rst in proto tcp from 10.0.0.0/8 to any

When sending back a TCP RST packet, IPFilter must construct a new packet that has the source address of the intended target, not the source address of the system it is running on (if they are different.)

For all of the other protocols handled by the IP protocol suite, to send back an error indicating that the received packet was dropped requires sending back an ICMP error packet. Whilst these can also be used for TCP, the sending host may not treat the received ICMP error as a hard error in(the same way as it does the TCP RST packet. To return an ICMP error); it is necessary to place return-icmp after the block keyword like this:

block return-icmp in proto udp from any to 192.168.0.1/24

When(electing to return an ICMP error packet, it is also possible to); select what type of ICMP error is returned. Whilst the full compliment of ICMP unreachable codes can be used by specifying a number instead of the string below, only the following should be used in conjunction with return-icmp.(Which return code to use is a choice to be made when); weighing up the pro's and con's. Using some of the codes may make it more obvious that a firewall is being used rather than just the host not responding.

filter-prohib (prohibited by filter) sending packets to the destination given in the received packet is prohibited due to the application of a packet filter

net-prohib (prohibited network) sending packets to the destination given in the received packet is administratively prohibited.

host-unk (host unknown) the destination host address is not known by the system receiving the packet and therefore cannot be reached.

host-unr (host unreachable) it is not possible to reach the host as given by the destination address in the packet header.

net-unk (network unknown) the destination network address is not known by the system receiving the packet and therefore cannot be reached.

net-unr (network unreachable) it is not possible to forward the packet on to its final destination as given by the destination address

port-unr (port unreachable) there is no application using the given destination port and therefore it is not possible to reach that port.

proto-unr (protocol unreachable) the IP protocol specified in the packet is not available to receive packets.

An example that shows how to send back a port unreachable packet for UDP packets to 192.168.1.0/24 is as follows:

block return-icmp(port-unr) in proto udp from any to 192.168.1.0/24

In the above examples, when sending the ICMP packet, IPFilter will construct a new ICMP packet with a source address of the network interface used to send the packet back to the original source. This can give away that there is an intermediate system blocking packets. To have IPFilter send back ICMP packets where the source address is the original destination, regardless of whether or not it is on the local host, return-icmp-as-dest is used like this:

```
block return-icmp-as-dest(port-unr) in proto udp \
   from any to 192.168.1.0/24
```

TCP/UDP Port Matching

Having specified which protocol is being matched, it is then possible to indicate which port numbers a packet must have in order to match the rule. Due to port numbers being used differently to addresses, it is therefore possible to match on them in different ways. IPFilter allows you to use the following logical operations:

< x is true if the port number is greater than or equal to x and less than or equal to y is true if the port number in the packet is less than x

<= x
is true if the port number in the packet is less than or equal to x

> x is true if the port number in the packet is greater than x

>= x
is true if the port number in the packet is greater or equal to x

= x is true if the port number in the packet is equal to x

!= x
is true if the port number in the packet is not equal to x

Additionally, there are a number of ways to specify a range of ports:

x <> y
is true if the port number is less than a and greater than y

x >< y
is true if the port number is greater than x and less than y

x:y is true if the port number is greater than or equal to x and less than or equal to y

Some examples of this are:

block in proto tcp from any port >= 1024 to any port < 1024
pass in proto tcp from 10.1.0.0/24 to any port = 22
block out proto udp from any to 10.1.1.1 port = 135
pass in proto udp from 1.1.1.1 port = 123 to 10.1.1.1 port = 123
pass in proto tcp from 127.0.0.0/8 to any port 6000:6009

If there is no desire to mention any specific source or destintion information in a filter rule then the word "all" can be used to indicate that all addresses are considered to match the rule.

IPv4 or IPv6

If a filter rule is constructed without any addresses then IPFilter will attempt to match both IPv4 and IPv6 packets with it. In the next list of rules, each one can be applied to either network protocol because there is no address specified from which IPFilter can derive with network protocol to expect.

pass in proto udp from any to any port = 53
block in proto tcp from any port < 1024 to any

To explicitly match a particular network address family with a specific rule, the family must be added to the rule. For IPv4 it is necessary to add family inet and for IPv6, family inet6. Thus the next rule will block all packets (both IPv4 and IPv6:

block in all

but in the following example, we block all IPv4 packets and only allow in IPv6 packets:

block in family inet all
pass in family inet6 all

To continue on from the example where we allowed either IPv4 or IPv6 packets to port 53 in, to change that such that only IPv6 packets to port 53 need to allowed blocked then it is possible to add in a protocol family qualifier:

pass in family inet6 proto udp from any to any port = 53

First match vs last match

To change the default behaviour from being the last matched rule decides the outcome to being the first matched rule, the word "quick" is inserted to the rule.

Extended Packet Matching

Beyond using plain addresses

On firewalls that are working with large numbers of hosts and networks or simply trying to filter discretely against various hosts, it can be an easier administration task to define a pool of addresses and have a filter rule reference that address pool rather than have a rule for each address.

In addition to being able to use address pools, it is possible to use the interface name(s) in the from/to address fields of a rule. If the name being used in the address section can be matched to any of the interface names mentioned in the rule's "on" or "via" fields then it can be used with one of the following keywords for extended effect:

broadcast use the primary broadcast address of the network interface for matching packets with this filter rule;

pass in on fxp0 proto udp from any to fxp0/broadcast port = 123

peer use the peer address on point to point network interfaces for matching packets with this filter rule. This option typically only has meaningful use with link protocols such as SLIP and PPP. For example, this rule allows ICMP packets from the remote peer of ppp0 to be received if they're destined for the address assigned to the link at the firewall end.

pass in on ppp0 proto icmp from ppp0/peer to ppp0/32

netmasked use the primary network address, with its netmask, of the network interface for matching packets with this filter rule. If a network interface had an IP address of 192.168.1.1 and its netmask was 255.255.255.0 (/24), then using the word "netmasked" after the interface name would match any addresses that would match 192.168.1.0/24. If we assume that bge0 has this IP address and netmask then the following two rules both serve to produce the same effect:

pass in on bge0 proto icmp from any to 192.168.1.0/24
pass in on bge0 proto icmp from any to bge0/netmasked

network using the primary network address, and its netmask, of the network interface, construct an address for exact matching. If a network interface has an address of 192.168.1.1 and its netmask is 255.255.255.0, using this option would only match packets to 192.168.1.0.

pass in on bge0 proto icmp from any to bge0/network

Another way to use the name of a network interface to get the address is to wrap the name in ()’s. In the above method, IPFilter looks at the interface names in use and to decide whether or not the name given is a hostname or network interface name. With the use of ()’s, it is possible to tell IPFilter that the name should be treated as a network interface name even though it doesn’t appear in the list of network interface that the rule ias associated with.

pass in proto icmp from any to (bge0)/32

Using address pools

Rather than list out multiple rules that either allow or deny specific addresses, it is possible to create a single object, call an address pool, that contains all of those addresses and reference that in the filter rule. For documentation on how to write the configuration file for those pools and load them, see ippool.conf(5) and ippool(8). There are two types of address pools that can be defined in ippool.conf(5): trees and hash tables. To refer to a tree defined in ippool.conf(5), use this syntax:

pass in from pool/trusted to any

Either a name or number can be used after the ’/’, just so long as it matches up with something that has already been defined in ipool.conf(5) and loaded in with ippool(8). Loading a filter rule that references a pool that does not exist will result in an error.

If hash tables have been used in ippool.conf(5) to store the addresses in instead of a tree, then replace the word pool with hash:

pass in from any to hash/webservers

There are different operational characteristics with each, so there may be some situations where a pool works better than hash and vice versa.

Matching TCP flags

The TCP header contains a field of flags that is used to decide if the packet is a connection request, connection termination, data, etc. By matching on the flags in conjunction with port numbers, it is possible to restrict the way in which IPFilter allows connections to be created. A quick overview of the TCP flags is below. Each is listed with the letter used in IPFilter rules, followed by its three or four letter pneumonic.

S SYN - this bit is set when a host is setting up a connection. The initiator typically sends a packet with the SYN bit and the responder sends back SYN plus ACK.

A ACK - this bit is set when the sender wishes to acknowledge the receipt of a packet from another host

P PUSH - this bit is set when a sending host has send some data that is yet to be acknowledged and a reply is sought

F FIN - this bit is set when one end of a connection starts to close the connection down

U URG - this bit is set to indicate that the packet contains urgent data

R RST - this bit is set only in packets that are a reply to another that has been received but is not targetted at any open port

C CWN

E ECN

When matching TCP flags, it is normal to just list the flag that you wish to be set. By default the set of flags it is compared against is "FSRPAU". Rules that say "flags S" will be displayed by ipfstat(8) as having "flags S/FSRPAU". This is normal. The last two flags, "C" and "E", are optional - they may or may not be used by an end host and have no bearing on either the acceptance of data nor control of the connection. Masking them out with "flags S/FSRPAUCE" may cause problems for remote hosts making a successful connection.

```
pass in quick proto tcp from any to any port = 22 flags S/SAFR
pass out quick proto tcp from any port = 22 to any flags SA
```

By itself, filtering based on the TCP flags becomes more work but when combined with stateful filtering (see below), the situation changes.

Matching on ICMP header information

The TCP and UDP are not the only protocols for which filtering beyond just the IP header is possible, extended matching on ICMP packets is also available. The list of valid ICMP types is different for IPv4 vs IPv6.

As a practical example, to allow the ping command to work against a specific target requires allowing two different types of ICMP packets, like this:

pass in proto icmp from any to webserver icmp-type echo
pass out proto icmp from webserver to any icmp-type echorep

The ICMP header has two fields that are of interest for filtering: the ICMP type and code. Filter rules can accept either a name or number for both the type and code. The list of names supported for ICMP types is listed below, however only ICMP unreachable errors have named codes (see above.)

The list of ICMP types that are available for matching an IPv4 packet are as follows:

echo (echo request), echorep (echo reply), inforeq (information request), inforep (information reply), maskreq (mask request), maskrep (mask reply), paramprob (parameter problem), redir (redirect), routerad (router advertisement), routersol (router solicit), squence (source quence), timest (timestamp), timestreq (timestamp reply), timex (time exceeded), unreach (unreachable).

The list of ICMP types that are available for matching an IPv6 packet are as follows:

echo (echo request), echorep (echo reply), fqdnquery (FQDN query), fqdnreply (FQDN reply), inforeq (information request), inforep (information reply), listendone (MLD listener done), listendqry (MLD listener query), listendrep (MLD listener reply), neighadvert (neighbour advert), neighborsol (neighbour solicit), paramprob (parameter problem), redir (redirect), renumber (router renumbering), routerad (router advertisement), routersol (router solicit), timex (time exceeded), toobig (packet too big), unreach (unreachable, whoreq (WRU request), whorep (WRU reply).

Stateful Packet Filtering

Stateful packet filtering is where IPFilter remembers some information from one or more packets that it has seen and is able to apply it to future packets that it receives from the network.

What this means for each transport layer protocol is different. For TCP it means that if IPFilter sees the very first packet of an attempt to make a connection, it has enough information to allow all other

subsequent packets without there needing to be any explicit rules to match them. IPFilter uses the TCP port numbers, TCP flags, window size and sequence numbers to determine which packets should be matched. For UDP, only the UDP port numbers are available. For ICMP, the ICMP types can be combined with the ICMP id field to determine which reply packets match a request/query that has already been seen. For all other protocols, only matching on IP address and protocol number is available for determining if a packet received is a mate to one that has already been let through.

The difference this makes is a reduction in the number of rules from 2 or 4 to 1. For example, these 4 rules:

```
pass in on bge0 proto tcp from any to any port = 22
pass out on bge1 proto tcp from any to any port = 22
pass in on bge1 proto tcp from any port = 22 to any
pass out on bge0 proto tcp from any port = 22 to any
```

can be replaced with this single rule:

```
pass in on bge0 proto tcp from any to any port = 22 flags S keep state
```

Similar rules for UDP and ICMP might be:

```
pass in on bge0 proto udp from any to any port = 53 keep state
pass in on bge0 proto icmp all icmp-type echo keep state
```

When using stateful filtering with TCP it is best to add "flags S" to the rule to ensure that state is only created when a packet is seen that is an indication of a new connection. Although IPFilter can gather some information from packets in the middle of a TCP connection to do stateful filtering, there are some options that are only sent at the start of the connection which alter the valid window of what TCP accepts. The end result of trying to pickup TCP state in mid connection is that some later packets that are part of the connection may not match the known state information and be dropped or blocked, causing problems. If a TCP packet matches IP addresses and port numbers but does not fit into the recognised window, it will not be automatically allowed and will be flagged inside of IPFitler as "out of window" (oow). See below, "Extra packet attributes", for how to match on this attribute.

Once a TCP connection has reached the established state, the default timeout allows for it to be idle for 5 days before it is removed from the state table. The timeouts for the other TCP connection states vary from 240 seconds to 30 seconds. Both UDP and ICMP state entries have asymetric timeouts where the timeout set upon seeing packets in the forward direction is much larger than for the reverse direction. For UDP the default timeouts are 120 and 12 seconds, for ICMP 60 and 6 seconds. This is a reflection of the use of these protocols being more for query-response than for ongoing connections. For all other

protocols the timeout is 60 seconds in both directions.

Stateful filtering options

The following options can be used with stateful filtering:

limit limit the number of state table entries that this rule can create to the number given after limit. A rule that has a limit specified is always permitted that many state table entries, even if creating an additional entry would cause the table to have more entries than the otherwise global limit.

```
pass ... keep state(limit 100)
```

age sets the timeout for the state entry when it sees packets going through it. Additionally it is possible to set the tieout for the reply packets that come back through the firewall to a different value than for the forward path. allowing a short timeout to be set after the reply has been seen and the state no longer required.

```
pass in quick proto icmp all icmp-type echo \
  keep state (age 3)
pass in quick proto udp from any \
  to any port = 53 keep state (age 30/1)
```

strict only has an impact when used with TCP. It forces all packets that are allowed through the firewall to be sequential: no out of order delivery of packets is allowed. This can cause significant slowdown for some connections and may stall others. Use with caution.

```
pass in proto tcp ... keep state(strict)
```

noicmperr prevents ICMP error packets from being able to match state table entries created with this flag using the contents of the original packet included.

```
pass ... keep state(noicmperr)
```

sync indicates to IPFilter that it needs to provide information to the user land daemons responsible for syncing other machines state tables up with this one.

```
pass ... keep state(sync)
```

nolog do not generate any log records for the creation or deletion of state table entries.

```
pass ... keep state(nolog)
```

icmp-head rather than just precent ICMP error packets from being able to match state table entries, allow an ACL to be processed that can filter in or out ICMP error packets based as you would with normal firewall rules. The icmp-head option requires a filter rule group number or name to be present, just as you would use with head.

pass in quick proto tcp ... keep state(icmp-head 101)
block in proto icmp from 10.0.0.0/8 to any group 101

max-srcs allows the number of distinct hosts that can create a state entry to be defined.

pass ... keep state(max-srcs 100)
pass ... keep state(limit 1000, max-srcs 100)

max-per-src whilst max-srcs limits the number of individual hosts that may cause the creation of a state table entry, each one of those hosts is still table to fill up the state table with new entries until the global maximum is reached. This option allows the number of state table entries per address to be limited.

pass ... keep state(max-srcs 100, max-per-src 1)
pass ... keep state(limit 100, max-srcs 100, max-per-src 1)

Whilst these two rules might seem identical, in that they both ultimately limit the number of hosts and state table entries created from the rule to 100, there is a subtle difference: the second will always allow up to 100 state table entries to be created whereas the first may not if the state table fills up from other rules.

Further, it is possible to specify a netmask size after the per-host limit that enables the per-host limit to become a per-subnet or per-network limit.

pass ... keep state(max-srcs 100, max-per-src 1/24)

If there is no IP protocol implied by addresses or other features of the rule, IPFilter will assume that no netmask is an all ones netmask for both IPv4 and IPv6.

Tieing down a connection

For any connection that transits a firewall, each packet will be seen twice: once going in and once going out. Thus a connection has 4 flows of packets:

forward inbound packets

forward outbound packets

reverse inbound packets

reverse outbound packets

IPFilter allows you to define the network interface to be used at all four points in the flow of packets. For rules that match inbound packets, out-via is used to specify which interfaces the packets go out, For rules that match outbound packets, in-via is used to match the inbound packets. In each case, the syntax generalises to this:

```
pass ... in on forward-in,reverse-in \
      out-via forward-out,reverse-out ...

pass ... out on forward-out,reverse-out \
       in-via forward-in,reverse-in ...
```

An example that pins down all 4 network interfaces used by an ssh connection might look something like this:

```
pass in on bge0,bge1 out-via bge1,bge0 proto tcp \
   from any to any port = 22 flags S keep state
```

Working with packet fragments

Fragmented packets result in 1 packet containing all of the layer 3 and 4 header information whilst the data is split across a number of other packets.

To enforce access control on fragmented packets, one of two approaches can be taken. The first is to allow through all of the data fragments (those that made up the body of the original packet) and rely on matching the header information in the "first" fragment, when it is seen. The reception of body fragments without the first will result in the receiving host being unable to completely reassemble the packet and discarding all of the fragments. The following three rules deny all fragmented packets from being received except those that are UDP and even then only allows those destined for port 2049 to be completed.

```
block in all with frags
pass in proto udp from any to any with frag-body
pass in proto udp from any to any port = 2049 with frags
```

Another mechanism that is available is to track "fragment state". This relies on the first fragment of a

packet that arrives to be the fragment that contains all of the layer 3 and layer 4 header information. With the receipt of that fragment before any other, it is possible to determine which other fragments are to be allowed through without needing to explicitly allow all fragment body packets. An example of how this is done is as follows:

pass in proto udp from any port = 2049 to any with frags keep frags

Building a tree of rules

Writing your filter rules as one long list of rules can be both inefficient in terms of processing the rules and difficult to understand. To make the construction of filter rules easier, it is possible to place them in groups. A rule can be both a member of a group and the head of a new group.

Using filter groups requires at least two rules: one to be in the group one one to send matchign packets to the group. If a packet matches a filtre rule that is a group head but does not match any of the rules in that group, then the packet is considered to have matched the head rule.

Rules that are a member of a group contain the word group followed by either a name or number that defines which group they’re in. Rules that form the branch point or starting point for the group must use the word head, followed by either a group name or number. If rules are loaded in that define a group but there is no matching head then they will effectively be orphaned rules. It is possible to have more than one head rule point to the same group, allowing groups to be used like subroutines to implement specific common policies.

A common use of filter groups is to define head rules that exist in the filter "main line" for each direction with the interfaces in use. For example:

block in quick on bge0 all head 100
block out quick on bge0 all head 101
block in quick on fxp0 all head internal-in
block out quick on fxp0 all head internal-out
pass in quick proto icmp all icmp-type echo group 100

In the above set of rules, there are four groups defined but only one of them has a member rule. The only packets that would be allowed through the above ruleset would be ICMP echo packets that are received on bge0.

Rules can be both a member of a group and the head of a new group, allowing groups to specialise.

block in quick on bge0 all head 100
block in quick proto tcp all head 1006 group 100

Another use of filter rule groups is to provide a place for rules to be dynamically added without needing to worry about their specific ordering amongst the entire ruleset. For example, if I was using this simple ruleset:

```
block in quick all with bad
block in proto tcp from any to any port = smtp head spammers
pass in quick proto tcp from any to any port = smtp flags S keep state
```

and I was getting lots of connections to my email server from 10.1.1.1 to deliver spam, I could load the following rule to complement the above:

```
block in quick from 10.1.1.1 to any group spammers
```

Decapsulation

Rule groups also form a different but vital role for decapsulation rules. With the following simple rule, if IPFilter receives an IP packet that has an AH header as its layer 4 payload, IPFilter would adjust its view of the packet internally and then jump to group 1001 using the data beyond the AH header as the new transport header.

```
decapsulate in proto ah all head 1001
```

For protocols that are recognised as being used with tunnelling or otherwise encapsulating IP protocols, IPFilter is able to decide what it has on the inside without any assistance. Some tunnelling protocols use UDP as the transport mechanism. In this case, it is necessary to instruct IPFilter as to what protocol is inside UDP.

```
decapsulate l5-as(ip) in proto udp from any \
   to any port = 1520 head 1001
```

Currently IPFilter only supports finding IPv4 and IPv6 headers directly after the UDP header.

If a packet matches a decapsulate rule but fails to match any of the rules that are within the specified group, processing of the packet continues to the next rule after the decapsulate and IPFilter's internal view of the packet is returned to what it was prior to the decapsulate rule.

It is possible to construct a decapsulate rule without the group head at the end that ipf(8) will accept but such rules will not result in anything happening.

Policy Based Routing

With firewalls being in the position they often are, at the boundary of different networks connecting

together and multiple connections that have different properties, it is often desirable to have packets flow in a direction different to what the routing table instructs the kernel. These decisions can often be extended to changing the route based on both source and destination address or even port numbers.

To support this kind of configuration, IPFilter allows the next hop destination to be specified with a filter rule. The next hop is given with the interface name to use for output. The syntax for this is interface:ip.address. It is expected that the address given as the next hop is directly connected to the network to which the interface is.

```
pass in on bge0 to bge1:1.1.1.1 proto tcp \
   from 1.1.2.3 to any port = 80 flags S keep state
```

When this feature is combined with stateful filtering, it becomes possible to influence the network interface used to transmit packets in both directions because we now have a sense for what its reverse flow of packets is.

```
pass in on bge0 to bge1:1.1.1.1 reply-to hme1:2.1.1.2 \
   proto tcp from 1.1.2.3 to any port = 80 flags S keep state
```

If the actions of the routing table are perfectly acceptable, but you would like to mask the presence of the firewall by not changing the TTL in IP packets as they transit it, IPFilter can be instructed to do a "fastroute" action like this:

```
pass in on bge0 fastroute proto icmp all
```

This should be used with caution as it can lead to endless packet loops. Additionally, policy based routing does not change the IP header’s TTL value.

A variation on this type of rule supports a duplicate of the original packet being created and sent out a different network. This can be useful for monitoring traffic and other purposes.

```
pass in on bge0 to bge1:1.1.1.1 reply-to hme1:2.1.1.2 \
   dup-to fxp0:10.0.0.1 proto tcp from 1.1.2.3 \
   to any port = 80 flags S keep state
```

Matching IPv4 options

The design for IPv4 allows for the header to be upto 64 bytes long, however most traffic only uses the basic header which is 20 bytes long. The other 44 bytes can be uesd to store IP options. These options are generally not necessary for proper interaction and function on the Internet today. For most people it is sufficient to block and drop all packets that have any options set. This can be achieved with this rule:

block in quick all with ipopts

This rule is usually placed towards the top of the configuration so that all incoming packets are blocked.

If you wanted to allow in a specific IP option type, the syntax changes slightly:

pass in quick proto igmp all with opt rtralrt

The following is a list of IP options that most people encounter and what their use/threat is.

lsrr (loose source route) the sender of the packet includes a list of addresses that they wish the packet to be routed through to on the way to the destination. Because replies to such packets are expected to use the list of addresses in reverse, hackers are able to very effectively use this header option in address spoofing attacks.

rr (record route) the sender allocates some buffer space for recording the IP address of each router that the packet goes through. This is most often used with ping, where the ping response contains a copy of all addresses from the original packet, telling the sender what route the packet took to get there. Due to performance and security issues with IP header options, this is almost no longer used.

rtralrt (router alert) this option is often used in IGMP messages as a flag to routers that the packet needs to be handled differently. It is unlikely to ever be received from an unknown sender. It may be found on LANs or otherwise controlled networks where the RSVP protocol and multicast traffic is in heavy use.

ssrr (strict source route) the sender of the packet includes a list of addresses that they wish the packet to be routed through to on the way to the destination. Where the lsrr option allows the sender to specify only some of the nodes the packet must go through, with the ssrr option, every next hop router must be specified.

The complete list of IPv4 options that can be matched on is: addext (Address Extention), cipso (Classical IP Security Option), dps (Dynamic Packet State), e-sec (Extended Security), eip (Extended Internet Protocol), encode (ENCODE), finn (Experimental Flow Control), imitd (IMI Traffic Descriptor), lsrr (Loose Source Route), mtup (MTU Probe - obsolete), mtur (MTU response - obsolete), nop (No Operation), nsapa (NSAP Address), rr (Record Route), rtralrt (Router Alert), satid (Stream Identifier), sdb (Selective Directed Broadcast), sec (Security), ssrr (Strict Source Route), tr (Tracerote), ts (Timestamp), ump (Upstream Multicast Packet), visa (Experimental Access Control) and zsu (Experimental Measurement).

Security with CIPSO and IPSO

IPFilter supports filtering on IPv4 packets using security attributes embedded in the IP options part of the packet. These options are usually only used on networks and systems that are using lablled security. Unless you know that you are using labelled security and your networking is also labelled, it is highly unlikely that this section will be relevant to you.

With the traditional IP Security Options (IPSO), packets can be tagged with a security level. The following keywords are recognised and match with the relevant RFC with respect to the bit patterns matched: confid (confidential), rserve-1 (1st reserved value), rserve-2 (2nd reserved value), rserve-3 (3rd reserved value), rserve-4 (4th reserved value), secret (secret), topsecret (top secret), unclass (unclassified).

```
block in quick all with opt sec-class unclass
pass in all with opt sec-class secret
```

Matching IPv6 extension headers

Just as it is possible to filter on the various IPv4 header options, so too it is possible to filter on the IPv6 extension headers that are placed between the IPv6 header and the transport protocol header.

dstopts (destination options), esp (encrypted, secure, payload), frag (fragment), hopopts (hop-by-hop options), ipv6 (IPv6 header), mobility (IP mobility), none, routing.

Logging

There are two ways in which packets can be logged with IPFilter. The first is with a rule that specifically says log these types of packets and the second is a qualifier to one of the other keywords. Thus it is possible to both log and allow or deny a packet with a single rule.

```
pass in log quick proto tcp from any to any port = 22
```

When using stateful filtering, the log action becomes part of the result that is remembered about a packet. Thus if the above rule was qualified with keep state, every packet in the connection would be logged. To only log the first packet from every packet flow tracked with keep state, it is necessary to indicate to IPFilter that you only wish to log the first packet.

```
pass in log first quick proto tcp from any to any port = 22 \
   flags S keep state
```

If it is a requirement that the logging provide an accurate representation of which connections are allowed, the log action can be qualified with the option or-block. This allows the administrator to instruct IPFilter to block the packet if the attempt to record the packet in IPFilter’s kernel log records

(which have an upper bound on size) failed. Unless the system shuts down or reboots, once a log record is written into the kernel buffer, it is there until ipmon(8) reads it.

```
block in log proto tcp from any to any port = smtp
pass in log or-block first quick proto tcp from any \
   to any port = 22 flags S keep state
```

By default, IPFilter will only log the header portion of a packet received on the network. Up to 128 bytes of a packet's body can also be logged with the body keyword. ipmon(8) will display the contents of the portion of the body logged in hex.

```
block in log body proto icmp all
```

When logging packets from ipmon(8) to syslog, by default ipmon(8) will control what syslog facility and priority a packet will be logged with. This can be tuned on a per rule basis like this:

```
block in quick log level err all with bad
pass in log level local1.info proto tcp \
   from any to any port = 22 flags S keep state
```

ipfstat(8) reports how many packets have been successfully logged and how many failed attempts to log a packet there were.

Filter rule comments

If there is a desire to associate a text string, be it an administrative comment or otherwise, with an IPFilter rule, this can be achieved by giving the filter rule a comment. The comment is loaded with the rule into the kernel and can be seen when the rules are listed with ipfstat.

```
pass in quick proto tcp from any \
   to port = 80 comment "all web server traffic is ok"
pass out quick proto tcp from any port = 80 \
   to any comment "all web server traffic is ok"
```

Tags

To enable filtering and NAT to correctly match up packets with rules, tags can be added at with NAT (for inbound packets) and filtering (for outbound packets.) This allows a filter to be correctly mated with its NAT rule in the event that the NAT rule changed the packet in a way that would mean it is not obvious what it was.

For inbound packets, IPFilter can match the tag used in the filter rules with that set by NAT. For

outbound rules, it is the reverse, the filter sets the tag and the NAT rule matches up with it.

```
pass in ... match-tag(nat=proxy)
pass out ... set-tag(nat=proxy)
```

Another use of tags is to supply a number that is only used with logging. When packets match these rules, the log tag is carried over into the log file records generated by ipmon(8). With the correct use of tools such as grep, extracting log records of interest is simplified.

```
block in quick log ... set-tag(log=33)
```

Filter Rule Expiration

IPFilter allows rules to be added into the kernel that it will remove after a specific period of time by specifying rule-ttl at the end of a rule. When listing rules in the kernel using ipfstat(8), rules that are going to expire will NOT display "rule-ttl" with the timeout, rather what will be seen is a comment with how many ipfilter ticks left the rule has to live.

The time to live is specified in seconds.

```
pass in on fxp0 proto tcp from any \
   to port = 22 flags S keep state rule-ttl 30
```

Internal packet attributes

In addition to being able to filter on very specific network and transport header fields, it is possible to filter on other attributes that IPFilter attaches to a packet. These attributes are placed in a rule after the keyword "with", as can be seen with frags and frag-body above. The following is a list of the other attributes available:

oow the packet’s IP addresses and TCP ports match an existing entry in the state table but the sequence numbers indicate that it is outside of the accepted window.

```
block return-rst in quick proto tcp from any to any with not oow
```

bcast this is set by IPFilter when it receives notification that the link layer packet was a broadcast packet. No checking of the IP addresses is performned to determine if it is a broadcast destination or not.

```
block in quick proto udp all with bcast
```

mcast this is set by IPFilter when it receives notification that the link layer packet was a multicast

packet. No checking of the IP addresses is performned to determine if it is a multicast destination or not.

pass in quick proto udp from any to any port = dns with mcast

mbcast can be used to match a packet that is either a multicast or broadcast packet at the link layer, as indicated by the operating system.

pass in quick proto udp from any to any port = ntp with mbcast

nat the packet positively matched a NAT table entry.

bad sanity checking of the packet failed. This could indicate that the layer 3/4 headers are not properly formed.

bad-src when reverse path verification is enabled, this flag will be set when the interface the packet is received on does not match that which would be used to send a packet out of to the source address in the received packet.

bad-nat an attempt to perform NAT on the packet failed.

not each one of the attributes matched using the "with" keyword can also be looked for to not be present. For example, to only allow in good packets, I can do this:

block in all
pass in all with not bad

Tuning IPFilter

The ipf.conf file can also be used to tune the behaviour of IPFilter, allowing, for example, timeouts for the NAT/state table(s) to be set along with their sizes. The presence and names of tunables may change from one release of IPFilter to the next. The tunables that can be changed via ipf.conf is the same as those that can be seen and modified using the -T command line option to ipf(8).

NOTE: When parsing ipf.conf, ipf(8) will apply the settings before loading any rules. Thus if your settings are at the top, these may be applied whilst the rules not applied if there is an error further down in the configuration file.

To set one of the values below, the syntax is simple: "set", followed by the name of the tuneable to set and then the value to set it to.

```
set state_max 9999;
set state_size 10101;
```

A list of the currently available variables inside IPFilter that may be tuned from ipf.conf are as follows:

active set through -s command line switch of ipf(8). See ipf(8) for detals.

chksrc when set, enables reverse path verification on source addresses and for filters to match packets with bad-src attribute.

control_forwarding when set turns off kernel forwarding when IPFilter is disabled or unloaded.

default_pass the default policy - whether packets are blocked or passed, etc - is represented by the value of this variable. It is a bit field and the bits that can be set are found in <netinet/ip_fil.h>. It is not recommended to tune this value directly.

ftp_debug set the debugging level of the in-kernel FTP proxy. Debug messages will be printed to the system console.

ftp_forcepasv when set the FTP proxy must see a PASV/EPSV command before creating the state/NAT entries for the 227 response.

ftp_insecure when set the FTP proxy will not wait for a user to login before allowing data connections to be created.

ftp_pasvonly when set the proxy will not create state/NAT entries for when it sees either the PORT or EPRT command.

ftp_pasvrdr when enabled causes the FTP proxy to create very insecure NAT/state entries that will allow any connection between the client and server hosts when a 227 reply is seen. Use with extreme caution.

ftp_single_xfer when set the FTP proxy will only allow one data connection at a time.

hostmap_size sets the size of the hostmap table used by NAT to store address mappings for use with sticky rules.

icmp_ack_timeout default timeout used for ICMP NAT/state when a reply packet is seen for an ICMP state that already exists

icmp_minfragmtu sets the minimum MTU that is considered acceptable in an ICMP error before deciding it is a bad packet.

icmp_timeout default timeout used for ICMP NAT/state when the packet matches the rule

ip_timeout default timeout used for NAT/state entries that are not TCP/UDP/ICMP.

ipf_flags

ips_proxy_debug this sets the debugging level for the proxy support code. When enabled, debugging messages will be printed to the system console.

log_all when set it changes the behaviour of "log body" to log the entire packet rather than just the first 128 bytes.

log_size sets the size of the in-kernel log buffer in bytes.

log_suppress when set, IPFilter will check to see if the packet it is logging is similar to the one it previously logged and if so, increases the occurance count for that packet. The previously logged packet must not have yet been read by ipmon(8).

min_ttl is used to set the TTL value that packets below will be marked with the low-ttl attribute.

nat_doflush if set it will cause the NAT code to do a more aggressive flush of the NAT table at the next opportunity. Once the flush has been done, the value is reset to 0.

nat_lock this should only be changed using ipfs(8)

nat_logging when set, NAT will create log records that can be read from /dev/ipnat.

nat_maxbucket maximum number of entries allowed to exist in each NAT hash bucket. This prevents an attacker trying to load up the hash table with entries in a single bucket, reducing performance.

nat_rules_size size of the hash table to store map rules.

nat_table_max maximum number of entries allowed into the NAT table

nat_table_size size of the hash table used for NAT

nat_table_wm_high when the fill percentage of the NAT table exceeds this mark, more aggressive

flushing is enabled.

nat_table_wm_low this sets the percentage at which the NAT table's agressive flushing will turn itself off at.

rdr_rules_size size of the hash table to store rdr rules.

state_lock this should only be changed using ipfs(8)

state_logging when set, the stateful filtering will create log records that can be read from /dev/ipstate.

state_max maximum number of entries allowed into the state table

state_maxbucket maximum number of entries allowed to exist in each state hash bucket. This prevents an attacker trying to load up the hash table with entries in a single bucket, reducing performance.

state_size size of the hash table used for stateful filtering

state_wm_freq this controls how often the agressive flushing should be run once the state table exceeds state_wm_high in percentage full.

state_wm_high when the fill percentage of the state table exceeds this mark, more aggressive flushing is enabled.

state_wm_low this sets the percentage at which the state table's agressive flushing will turn itself off at.

tcp_close_wait timeout used when a TCP state entry reaches the FIN_WAIT_2 state.

tcp_closed timeout used when a TCP state entry is ready to be removed after either a RST packet is seen.

tcp_half_closed timeout used when a TCP state entry reaches the CLOSE_WAIT state.

tcp_idle_timeout timeout used when a TCP state entry reaches the ESTABLISHED state.

tcp_last_ack timeout used when a TCP NAT/state entry reaches the LAST_ACK state.

tcp_syn_received timeout applied to a TCP NAT/state entry after SYN-ACK packet has been seen.

tcp_syn_sent timeout applied to a TCP NAT/state entry after SYN packet has been seen.

tcp_time_wait timeout used when a TCP NAT/state entry reaches the TIME_WAIT state.

tcp_timeout timeout used when a TCP NAT/state entry reaches either the half established state (one ack is seen after a SYN-ACK) or one side is in FIN_WAIT_1.

udp_ack_timeout default timeout used for UDP NAT/state when a reply packet is seen for a UDP state that already exists

udp_timeout default timeout used for UDP NAT/state when the packet matches the rule

update_ipid when set, turns on changing the IP id field in NAT'd packets to a random number.

Table of visible variables

A list of all of the tunables, their minimum, maximum and current values is as follows.

Name		Min	Max	Current
active	0	0	0	
chksrc	0	1	0	
control_forwarding	0	1	0	
default_pass	0	MAXUINT		134217730
ftp_debug		0	10	0
ftp_forcepasv	0	1	1	
ftp_insecure	0	1	0	
ftp_pasvonly	0	1	0	
ftp_pasvrdr	0	1	0	
ftp_single_xfer	0	1	0	
hostmap_size	1	MAXINT		2047
icmp_ack_timeout	1	MAXINT		12
icmp_minfragmtu	0	1	68	
icmp_timeout	1	MAXINT		120
ip_timeout	1	MAXINT		120
ipf_flags		0	MAXUINT	0
ips_proxy_debug	0	10	0	
log_all	0	1	0	
log_size		0	524288	32768
log_suppress	0	1	1	
min_ttl	0	1	4	
nat_doflush	0	1	0	
nat_lock		0	1	0
nat_logging	0	1	1	

nat_maxbucket	1	MAXINT	22	
nat_rules_size	1	MAXINT	127	
nat_table_max	1	MAXINT	30000	
nat_table_size	1	MAXINT	2047	
nat_table_wm_high	2	100	99	
nat_table_wm_low	1	99	90	
rdr_rules_size	1	MAXINT	127	
state_lock	0	1	0	
state_logging	0	1	1	
state_max		1	MAXINT	4013
state_maxbucket	1	MAXINT	26	
state_size	1	MAXINT	5737	
state_wm_freq	2	999999	20	
state_wm_high	2	100	99	
state_wm_low	1	99	90	
tcp_close_wait	1	MAXINT	480	
tcp_closed	1	MAXINT	60	
tcp_half_closed	1	MAXINT	14400	
tcp_idle_timeout	1	MAXINT	864000	
tcp_last_ack	1	MAXINT	60	
tcp_syn_received	1	MAXINT	480	
tcp_syn_sent	1	MAXINT	480	
tcp_time_wait	1	MAXINT	480	
tcp_timeout	1	MAXINT	480	
udp_ack_timeout	1	MAXINT	24	
udp_timeout	1	MAXINT	240	
update_ipid	0	1	0	

Calling out to internal functions

IPFilter provides a pair of functions that can be called from a rule that allow for a single rule to jump out to a group rather than walk through a list of rules to find the group. If you've got multiple networks, each with its own group of rules, this feature may help provide better filtering performance.

The lookup to find which rule group to jump to is done on either the source address or the destination address but not both.

In this example below, we are blocking all packets by default but then doing a lookup on the source address from group 1010. The two rules in the ipf.conf section are lone members of their group. For an incoming packet that is from 1.1.1.1, it will go through three rules: (1) the block rule, (2) the call rule and (3) the pass rule for group 1020. For a packet that is from 3.3.2.2, it will also go through three

rules: (1) the block rule, (2) the call rule and (3) the pass rule for group 1030. Should a packet from 3.1.1.1 arrive, it will be blocked as it does not match any of the entries in group 1010, leaving it to only match the first rule.

```
from ipf.conf
-------------
block in all
call now srcgrpmap/1010 in all
pass in proto tcp from any to any port = 80 group 1020
pass in proto icmp all icmp-type echo group 1030

from ippool.conf
----------------
group-map in role=ipf number=1010
   { 1.1.1.1 group = 1020, 3.3.0.0/16 group = 1030; };
```

IPFilter matching expressions

An experimental feature that has been added to filter rules is to use the same expression matching that is available with various commands to flush and list state/NAT table entries. The use of such an expression precludes the filter rule from using the normal IP header matching.

```
pass in exp { "tcp.sport 23 or tcp.sport 50" } keep state
```

Filter rules with BPF

On platforms that have the BPF built into the kernel, IPFilter can be built to allow BPF expressions in filter rules. This allows for packet matching to be on arbitrary data in the packt. The use of a BPF expression replaces all of the other protocol header matching done by IPFilter.

```
pass in bpf-v4 { "tcp and (src port 23 or src port 50)" } \
   keep state
```

These rules tend to be write-only because the act of compiling the filter expression into the BPF instructions loaded into the kernel can make it difficut to accurately reconstruct the original text filter. The end result is that while ipf.conf() can be easy to read, understanding the output from ipfstat might not be.

VARIABLES

This configuration file, like all others used with IPFilter, supports the use of variable substitution throughout the text.

```
nif="ppp0";
pass in on $nif from any to any
```

would become

```
pass in on ppp0 from any to any
```

Variables can be used recursively, such as ’foo="$bar baz";’, so long as $bar exists when the parser reaches the assignment for foo.

See **ipf(8)** for instructions on how to define variables to be used from a shell environment.

FILES

/dev/ipf /etc/ipf.conf
/usr/share/examples/ipfilter Directory with examples.

SEE ALSO

ipf(8), ipfstat(8), ippool.conf(5), ippool(8)