NAME

ipnat, ipnat.conf - IPFilter NAT file format

DESCRIPTION

The **ipnat.conf** file is used to specify rules for the Network Address Translation (NAT) component of IPFilter. To load rules specified in the **ipnat.conf** file, the **ipnat(8)** program is used.

For standard NAT functionality, a rule should start with **map** and then proceeds to specify the interface for which outgoing packets will have their source address rewritten. Following this it is expected that the old source address, and optionally port number, will be specified.

In general, all NAT rules conform to the following layout: the first word indicates what type of NAT rule is present, this is followed by some stanzas to match a packet, followed by a "->" and this is then followed by several more stanzas describing the new data to be put in the packet.

In this text and in others, use of the term "left hand side" (LHS) when talking about a NAT rule refers to text that appears before the "->" and the "right hand side" (RHS) for text that appears after it. In essence, the LHS is the packet matching and the RHS is the new data to be used.

VARIABLES

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

```
nif="ppp0";
map $nif 0/0 -> 0/32
```

would become

```
map ppp0 0/0 -> 0/32
```

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

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

OUTBOUND SOURCE TRANSLATION (map’ing)

Changing the source address of a packet is traditionally performed using **map** rules. Both the source address and optionally port number can be changed according to various controls.

To start out with, a common rule used is of the form:

map le0 0/0 -> 0/32

Here we're saying change the source address of all packets going out of le0 (the address/mask pair of 0/0 matching all packets) to that of the interface le0 (0/32 is a synonym for the interface's own address at the current point in time.) If we wanted to pass the packet through with no change in address, we would write it as:

map le0 0/0 -> 0/0

If we only want to change a portion of our internal network and to a different address that is routed back through this host, we might do:

map le0 10.1.1.0/24 -> 192.168.55.3/32

In some instances, we may have an entire subnet to map internal addresses out onto, in which case we can express the translation as this:

map le0 10.0.0.0/8 -> 192.168.55.0/24

IPFilter will cycle through each of the 256 addresses in the 192.168.55.0/24 address space to ensure that they all get used.

Of course this poses a problem for TCP and UDP, with many connections made, each with its own port number pair. If we're unlucky, translations can be dropped because the new address/port pair mapping already exists. To mitigate this problem, we add in port translation or port mapping:

map le0 10.0.0.0/8 -> 192.168.55.0/24 portmap tcp/udp auto

In this instance, the word "auto" tells IPFilter to calculate a private range of port numbers for each address on the LHS to use without fear of them being trampled by others. This can lead to problems if there are connections being generated more quickly than IPFilter can expire them. In this instance, and if we want to get away from a private range of port numbers, we can say:

map le0 10.0.0.0/8 -> 192.168.55.0/24 portmap tcp/udp 5000:65000

And now each connection through le0 will add to the enumeration of the port number space 5000-65000 as well as the IP address subnet of 192.168.55.0/24.

If the new addresses to be used are in a consecutive range, rather than a complete subnet, we can express this as:

```
map le0 10.0.0.0/8 -> range 192.168.55.10-192.168.55.249
        portmap tcp/udp 5000:65000
```

This tells IPFilter that it has a range of 240 IP address to use, from 192.168.55.10 to 192.168.55.249, inclusive.

If there were several ranges of addresses for use, we can use each one in a round-robin fashion as followed:

```
map le0 10.0.0.0/8 -> range 192.168.55.10-192.168.55.29
        portmap tcp/udp 5000:65000 round-robin
map le0 10.0.0.0/8 -> range 192.168.55.40-192.168.55.49
        portmap tcp/udp 5000:65000 round-robin
```

To specify translation rules that impact a specific IP protocol, the protocol name or number is appended to the rule like this:

```
map le0 10.0.0.0/8 -> 192.168.55.0/24 tcp/udp
map le0 10.0.0.0/8 -> 192.168.55.1/32 icmp
map le0 10.0.0.0/8 -> 192.168.55.2/32 gre
```

For TCP connections exiting a connection such as PPPoE where the MTU is slightly smaller than normal ethernet, it can be useful to reduce the Maximum Segment Size (MSS) offered by the internal machines to match, reducing the liklihood that the either end will attempt to send packets that are too big and result in fragmentation. This is acheived using the **mssclamp** option with TCP **map** rules like this:

```
map pppoe0 0/0 -> 0/32 mssclamp 1400 tcp
```

For ICMP packets, we can map the ICMP id space in query packets:

```
map le0 10.0.0.0/8 -> 192.168.55.1/32 icmpidmap icmp 1000:20000
```

If we wish to be more specific about our initial matching criteria on the LHS, we can expand to using a syntax more similar to that in **ipf.conf(5)** :

```
map le0 from 10.0.0.0/8 to 26.0.0.0/8 ->
        192.168.55.1
map le0 from 10.0.0.0/8 port > 1024 to 26.0.0.0/8 ->
        192.168.55.2 portmap 5000:9999 tcp/udp
```

```
map le0 from 10.0.0.0/8 ! to 26.0.0.0/8 ->
                192.168.55.3 portmap 5000:9999 tcp/udp
```

NOTE:
negation matching with source addresses is **NOT** possible with **map** / **map-block** rules.

The NAT code has builtin default timeouts for TCP, UDP, ICMP and another for all other protocols. In general, the timeout for an entry to be deleted shrinks once a reply packet has been seen (excluding TCP.) If you wish to specify your own timeouts, this can be achieved either by setting one timeout for both directions:

```
map le0 0/0 -> 0/32 gre age 30
```

or setting a different timeout for the reply:

```
map le0 from any to any port = 53 -> 0/32 age 60/10 udp
```

A pressing problem that many people encounter when using NAT is that the address protocol can be embedded inside an application's communication. To address this problem, IPFilter provides a number of built-in proxies for the more common trouble makers, such as FTP. These proxies can be used as follows:

```
map le0 0/0 -> 0/32 proxy port 21 ftp/tcp
```

In this rule, the word "proxy" tells us that we want to connect up this translation with an internal proxy. The "port 21" is an extra restriction that requires the destination port number to be 21 if this rule is to be activated. The word "ftp" is the proxy identifier that the kernel will try and resolve internally, "tcp" the protocol that packets must match.

See below for a list of proxies and their relative staus.

To associate NAT rules with filtering rules, it is possible to set and match tags during either inbound or outbound processing. At present the tags for forwarded packets are not preserved by forwarding, so once the packet leaves IPFilter, the tag is forgotten. For **map** rules, we can match tags set by filter rules like this:

```
map le0 0/0 -> 0/32 proxy portmap 5000:5999 tag lan1 tcp
```

This would be used with "pass out" rules that includes a stanza such as "set-tag (nat = lan1)".

If the interface in which packets are received is different from the interface on which packets are sent out, then the translation rule needs to be written to take this into account:

map hme0,le0 0/0 -> 0/32

Although this might seem counterintuitive, the interfaces when listed in rules for **ipnat.conf** are always in the *inbound* , *outbound* order. In this case, hme0 would be the return interface and le0 would be the outgoing interface. If you wish to allow return packets on any interface, the correct syntax to use would be:

map *,le0 0/0 -> 0/32

A special variant of **map** rules exists, called **map-block.** This command is intended for use when there is a large network to be mapped onto a smaller network, where the difference in netmasks is upto 14 bits difference in size. This is achieved by dividing the address space and port space up to ensure that each source address has its own private range of ports to use. For example, this rule:

map-block ppp0 172.192.0.0/16 -> 209.1.2.0/24 ports auto

would result in 172.192.0.0/24 being mapped to 209.1.2.0/32 with each address, from 172.192.0.0 to 172.192.0.255 having 252 ports of its own. As opposed to the above use of **map**, if for some reason the user of (say) 172.192.0.2 wanted 260 simultaneous connections going out, they would be limited to 252 with **map-block** but would just *move on* to the next IP address with the **map** command.

Extended matching

If it is desirable to match on both the source and destination of a packet before applying an address translation to it, this can be achieved by using the same from-to syntax as is used in **ipf.conf**(5). What follows applies equally to the **map** rules discussed above and **rdr** rules discussed below. A simple example is as follows:

map bge0 from 10.1.0.0/16 to 192.168.1.0/24 -> 172.12.1.4

This would only match packets that are coming from hosts that have a source address matching 10.1.0.0/16 and a destination matching 192.168.1.0/24. This can be expanded upon with ports for TCP like this:

rdr bge0 from 10.1.0.0/16 to any port = 25 -> 127.0.0.1 port 2501 tcp

Where only TCP packets from 10.1.0.0/16 to port 25 will be redirected to port 2501.

As with **ipf.conf**(5), if we have a large set of networks or addresses that we would like to match up with then we can define a pool using **ippool**(8) in **ippool.conf**(5) and then refer to it in an **ipnat** rule like this:

map bge0 from pool/100 to any port = 25 -> 127.0.0.1 port 2501 tcp

NOTE:

In this situation, the rule is considered to have a netmask of "0" and thus is looked at last, after any rules with /16's or /24's in them, *even if* the defined pool only has /24's or /32's. Pools may also be used *wherever* the from-to syntax in **ipnat.conf**(5) is allowed.

INBOUND DESTINATION TRANSLATION (redirection)

Redirection of packets is used to change the destination fields in a packet and is supported for packets that are moving *in* on a network interface. While the same general syntax for **map** rules is supported, there are differences and limitations.

Firstly, by default all redirection rules target a single IP address, not a network or range of network addresses, so a rule written like this:

rdr le0 0/0 -> 192.168.1.0

Will not spread packets across all 256 IP addresses in that class C network. If you were to try a rule like this:

rdr le0 0/0 -> 192.168.1.0/24

then you will receive a parsing error.

The from-to source-destination matching used with **map** rules can be used with rdr rules, along with negation, however the restriction moves - only a source address match can be negated:

rdr le0 from 1.1.0.0/16 to any -> 192.168.1.3
rdr le0 ! from 1.1.0.0/16 to any -> 192.168.1.4

If there is a consective set of addresses you wish to spread the packets over, then this can be done in one of two ways, the word "range" optional to preserve:

rdr le0 0/0 -> 192.168.1.1 - 192.168.1.5
rdr le0 0/0 -> range 192.168.1.1 - 192.168.1.5

If there are only two addresses to split the packets across, the recommended method is to use a comma

(",") like this:

rdr le0 0/0 -> 192.168.1.1,192.168.1.2

If there is a large group of destination addresses that are somewhat disjoint in nature, we can cycle through them using a **round-robin** technique like this:

rdr le0 0/0 -> 192.168.1.1,192.168.1.2 round-robin
rdr le0 0/0 -> 192.168.1.5,192.168.1.7 round-robin
rdr le0 0/0 -> 192.168.1.9 round-robin

If there are a large number of redirect rules and hosts being targetted then it may be desirable to have all those from a single source address be targetted at the same destination address. To achieve this, the word **sticky** is appended to the rule like this:

rdr le0 0/0 -> 192.168.1.1,192.168.1.2 sticky
rdr le0 0/0 -> 192.168.1.5,192.168.1.7 round-robin sticky
rdr le0 0/0 -> 192.168.1.9 round-robin sticky

The **sticky** feature can only be combined with **round-robin** and the use of comma.

For TCP and UDP packets, it is possible to both match on the destiantion port number and to modify it. For example, to change the destination port from 80 to 3128, we would use a rule like this:

rdr de0 0/0 port 80 -> 127.0.0.1 port 3128 tcp

If a range of ports is given on the LHS and a single port is given on the RHS, the entire range of ports is moved. For example, if we had this:

rdr le0 0/0 port 80-88 -> 127.0.0.1 port 3128 tcp

then port 80 would become 3128, port 81 would become 3129, etc. If we want to redirect a number of different pots to just a single port, an equals sign ("=") is placed before the port number on the RHS like this:

rdr le0 0/0 port 80-88 -> 127.0.0.1 port = 3128 tcp

In this case, port 80 goes to 3128, port 81 to 3128, etc.

As with **map** rules, it is possible to manually set a timeout using the **age** option, like this:

```
rdr le0 0/0 port 53 -> 127.0.0.1 port 10053 udp age 5/5
```

The use of proxies is not restricted to **map** rules and outbound sessions. Proxies can also be used with redirect rules, although the syntax is slightly different:

```
rdr ge0 0/0 port 21 -> 127.0.0.1 port 21 tcp proxy ftp
```

For **rdr** rules, the interfaces supplied are in the same order as **map** rules - input first, then output. In situations where the outgoing interface is not certain, it is also possible to use a wildcard ("*") to effect a match on any interface.

```
rdr le0,* 0/0 -> 192.168.1.0
```

A single rule, with as many options set as possible would look something like this:

```
rdr le0,ppp0 9.8.7.6/32 port 80 -> 1.1.1.1,1.1.1.2 port 80 tcp
   round-robin frag age 40/40 sticky mssclamp 1000 tag tagged
```

REWRITING SOURCE AND DESTINATION

Whilst the above two commands provide a lot of flexibility in changing addressing fields in packets, often it can be of benefit to translate *both* source **and** destination at the same time or to change the source address on input or the destination address on output. Doing all of these things can be accomplished using **rewrite** NAT rules.

A **rewrite** rule requires the same level of packet matching as before, protocol and source/destination information but in addition allows either **in** or **out** to be specified like this:

```
rewrite in on ppp0 proto tcp from any to any port = 80 ->
        src 0/0 dst 127.0.0.1,3128;
rewrite out on ppp0 from any to any ->
        src 0/32 dst 10.1.1.0/24;
```

On the RHS we can specify both new source and destination information to place into the packet being sent out. As with other rules used in **ipnat.conf**, there are shortcuts syntaxes available to use the original address information (**0/0**) and the address associated with the network interface (**0/32**.) For TCP and UDP, both address and port information can be changed. At present it is only possible to specify either a range of port numbers to be used (**X-Y**) or a single port number (**= X**) as follows:

```
rewrite in on le0 proto tcp from any to any port = 80 ->
        src 0/0,2000-20000 dst 127.0.0.1,port = 3128;
```

There are four fields that are stepped through in enumerating the number space available for creating a new destination:

source address

source port

destination address

destination port

If one of these happens to be a static then it will be skipped and the next one incremented. As an example:

```
rewrite out on le0 proto tcp from any to any port = 80 ->
        src 1.0.0.0/8,5000-5999 dst 2.0.0.0/24,6000-6999;
```

The translated packets would be:

1st src=1.0.0.1,5000 dst=2.0.0.1,6000

2nd src=1.0.0.2,5000 dst=2.0.0.1,6000

3rd src=1.0.0.2,5001 dst=2.0.0.1,6000

4th src=1.0.0.2,5001 dst=2.0.0.2,6000

5th src=1.0.0.2,5001 dst=2.0.0.2,6001

6th src=1.0.0.3,5001 dst=2.0.0.2,6001

and so on.

As with **map** rules, it is possible to specify a range of addresses by including the word *range* before the addresses:

```
rewrite from any to any port = 80 ->
        src 1.1.2.3 - 1.1.2.6 dst 2.2.3.4 - 2.2.3.6;
```

DIVERTING PACKETS

If you'd like to send packets to a UDP socket rather than just another computer to be decapsulated, this can be achieved using a **divert** rule.

Divert rules can be used with both inbound and outbound packet matching however the rule **must** specify host addresses for the outer packet, not ranges of addresses or netmasks, just single addresses. Additionally the syntax must supply required information for UDP. An example of what a divert rule looks ike is as follows:

```
divert in on le0 proto udp from any to any port = 53 ->
        src 192.1.1.1,54 dst 192.168.1.22.1,5300;
```

On the LHS is a normal set of matching capabilities but on the RHS it is a requirement to specify both the source and destination addresses and ports.

As this feature is intended to be used with targetting packets at sockets and not IPFilter running on other systems, there is no rule provided to *undivert* packets.

NOTE:

Diverted packets *may* be fragmented if the addition of the encapsulating IP header plus UDP header causes the packet to exceed the size allowed by the outbound network interface. At present it is not possible to cause Path MTU discovery to happen as this feature is intended to be transparent to both endpoints. **Path MTU Discovery** If Path MTU discovery is being used and the "do not fragment" flag is set in packets to be encapsulated, an ICMP error message will be sent back to the sender if the new packet would need to be fragmented.

COMMON OPTIONS

This section deals with options that are available with all rules.

purge

When the purge keyword is added to the end of a NAT rule, it will cause all of the active NAT sessions to be removed when the rule is removed as an individual operation. If all of the NAT rules are flushed out, it is expected that the operator will similarly flush the NAT table and thus NAT sessions are not removed when the NAT rules are flushed out.

RULE ORDERING

NOTE: Rules in **ipnat.conf** are read in sequentially as listed and loaded into the kernel in this fashion **BUT** packet matching is done on **netmask**, going from 32 down to 0. If a rule uses **pool** or **hash** to reference a set of addresses or networks, the netmask value for these fields is considered to be "0". So if your **ipnat.conf** has the following rules:

```
rdr le0 192.0.0.0/8 port 80 -> 127.0.0.1 3132 tcp
rdr le0 192.2.0.0/16 port 80 -> 127.0.0.1 3131 tcp
rdr le0 from any to pool/100 port 80 -> 127.0.0.1 port 3130 tcp
rdr le0 192.2.2.0/24 port 80 -> 127.0.0.1 3129 tcp
rdr le0 192.2.2.1 port 80 -> 127.0.0.1 3128 tcp
```

then the rule with 192.2.2.1 will match **first**, regardless of where it appears in the ordering of the above rules. In fact, the order in which they would be used to match a packet is:

```
rdr le0 192.2.2.1 port 80 -> 127.0.0.1 3128 tcp
rdr le0 192.2.2.0/24 port 80 -> 127.0.0.1 3129 tcp
rdr le0 192.2.0.0/16 port 80 -> 127.0.0.1 3131 tcp
rdr le0 192.0.0.0/8 port 80 -> 127.0.0.1 3132 tcp
rdr le0 from any to pool/100 port 80 -> 127.0.0.1 port 3130 tcp
```

where the first line is actually a /32.

If your **ipnat.conf** file has entries with matching target fields (source address for **map** rules and destination address for **rdr** rules), then the ordering in the **ipnat.conf** file does matter. So if you had the following:

```
rdr le0 from 1.1.0.0/16 to 192.2.2.1 port 80 -> 127.0.0.1 3129 tcp
rdr le0 from 1.1.1.0/24 to 192.2.2.1 port 80 -> 127.0.0.1 3128 tcp
```

Then no packets will match the 2nd rule, they'll all match the first.

IPv6

In all of the examples above, where an IPv4 address is present, an IPv6 address can also be used. All rules must use either IPv4 addresses with both halves of the NAT rule or IPv6 addresses for both halves. Mixing IPv6 addresses with IPv4 addresses, in a single rule, will result in an error.

For shorthand notations such as "0/32", the equivalent for IPv6 is "0/128". IPFilter will treat any netmask greater than 32 as an implicit direction that the address should be IPv6, not IPv4. To be unambiguous with 0/0, for IPv6 use ::0/0.

KERNEL PROXIES

IP Filter comes with a few, simple, proxies built into the code that is loaded into the kernel to allow secondary channels to be opened without forcing the packets through a user program. The current state of the proxies is listed below, as one of three states:

Aging - protocol is roughly understood from the time at which the proxy was written but it is not well tested or maintained;

Developmental - basic functionality exists, works most of the time but may be problematic in extended real use;

Experimental - rough support for the protocol at best, may or may not work as testing has been at best sporadic, possible large scale changes to the code in order to properly support the protocol.

Mature - well tested, protocol is properly understood by the proxy;

The currently compiled in proxy list is as follows:

FTP - Mature
(map ... proxy port ftp ftp/tcp)

IRC - Experimental
(proxy port 6667 irc/tcp)

rpcbind - Experimental

PPTP - Experimental

H.323 - Experimental
(map ... proxy port 1720 h323/tcp)

Real Audio (PNA) - Aging

DNS - Developmental
(map ... proxy port 53 dns/udp { block .cnn.com; })

IPsec - Developmental
(map ... proxy port 500 ipsec/tcp)

netbios - Experimental

R-command - Mature
(map ... proxy port shell rcmd/tcp)

KERNEL PROXIES

FILES

/dev/ipnat
/etc/protocols
/etc/services
/etc/hosts

SEE ALSO

ipnat(4), hosts(5), ipf(5), services(5), ipf(8), ipnat(8)