nft

nft — Administration tool of the nftables framework for packet filtering and classification

Synopsis

`nft [ -nNscae ] [ -I directory] [ -f filename | -i | cmd]`

`nft -h`

`nft -v`

Description

nft is the command line tool used to set up, maintain and inspect packet filtering and classification rules in the Linux kernel, in the nftables framework. The Linux kernel subsystem is known as nf_tables, and 'nf' stands for Netfilter.

Options

For a full summary of options, run **nft --help**.

-h, --help Show help message and all options.

-v, --version Show version.

-n, --numeric Show data numerically. When used once (the default behaviour), skip lookup of addresses to symbolic names. Use twice to also show Internet services (port numbers) numerically. Use three times to also show protocols and UIDs/GIDs numerically.

-s, --stateless Omit stateful information of rules and stateful objects.

-l, --literal Translate numeric to literal. When used once (the default behaviour), print services (instead of numerical port numbers). Use twice to perform the IP address to name lookup, this usually requires network traffic for DNS lookup that slows down the ruleset listing.

-c, --check Check commands validity without actually applying the changes.

-a, --handle Show object handles in output.

-e, --echo When inserting items into the ruleset using **add**, **insert** or **replace** commands, print notifications just like **nft monitor**.

-I, --includepath *directory* Add the directory *directory* to the list of directories to be searched for included files. This option may be specified multiple times.

-f, --file *filename* Read input from *filename*. If *filename* is -, read from `stdin`.

nft scripts must start **#!/usr/sbin/nft -f**

-i, --interactive Read input from an interactive readline CLI. You can use **quit** to exit, or use the `EOF` marker, normally this is `CTRL-D`.

Input file format

Lexical conventions

Input is parsed line-wise. When the last character of a line, just before the newline character, is a non-quoted backslash (`\`), the next line is treated as a continuation. Multiple commands on the same line can be separated using a semicolon (`;`).

A hash sign (`#`) begins a comment. All following characters on the same line are ignored.

Identifiers begin with an alphabetic character (`a-z,A-Z`), followed zero or more alphanumeric characters (`a-z,A-Z,0-9`) and the characters slash (`/`), backslash (`\`), underscore (`_`) and dot (`.`). Identifiers using different characters or clashing with a keyword need to be enclosed in double quotes (`"`).

Include files

`include "`*`filename`*`"`

Other files can be included by using the **include** statement. The directories to be searched for include files can be specified using the `-I/--includepath` option. You can override this behaviour either by prepending ./ to your path to force inclusion of files located in the current working directory (i.e. relative path) or / for file location expressed as an absolute path.

If -I/--includepath is not specified, then nft relies on the default directory that is specified at compile time. You can retrieve this default directory via -h/--help option.

Include statements support the usual shell wildcard symbols (`*,?,[]`). Having no matches for an include statement is not an error, if wildcard symbols are used in the include statement. This allows having potentially empty include directories for statements like `include "/etc/firewall/rules/*"`. The wildcard matches are loaded in alphabetical order. Files beginning with dot (`.`) are not matched by include statements.

Symbolic variables

`define` *`variable`* `=` *`expr`*

`$`*`variable`*

Symbolic variables can be defined using the **define** statement. Variable references are expressions and can be used initialize other variables. The scope of a definition is the current block and all blocks contained within.

Example 1.1 Using symbolic variables

```
define int_if1 = eth0
define int_if2 = eth1
define int_ifs = { $int_if1, $int_if2 }

filter input iif $int_ifs accept
```

Address families

Address families determine the type of packets which are processed. For each address family the kernel contains so called hooks at specific stages of the packet processing paths, which invoke nftables if rules for these hooks exist.

ip IPv4 address family.

ip6 IPv6 address family.

inet Internet (IPv4/IPv6) address family.

arp ARP address family, handling IPv4 ARP packets.

bridge Bridge address family, handling packets which traverse a bridge device.

netdev Netdev address family, handling packets from ingress.

All nftables objects exist in address family specific namespaces, therefore all identifiers include an address family. If an identifier is specified without an address family, the `ip` family is used by default.

IPv4/IPv6/Inet address families

The IPv4/IPv6/Inet address families handle IPv4, IPv6 or both types of packets. They contain five hooks at different packet processing stages in the network stack.

Hook	Description
prerouting	All packets entering the system are processed by the prerouting hook. It is invoked before the routing process and is used for early filtering or changing packet attributes that affect routing.
input	Packets delivered to the local system are processed by the input hook.
forward	Packets forwarded to a different host are processed by the forward hook.
output	Packets sent by local processes are processed by the output hook.
postrouting	All packets leaving the system are processed by the postrouting hook.

Table 1: IPv4/IPv6/Inet address family hooks

ARP address family

The ARP address family handles ARP packets received and sent by the system. It is commonly used to mangle ARP packets for clustering.

Hook	Description
input	Packets delivered to the local system are processed by the input hook.
output	Packets send by the local system are processed by the output hook.

Table 2: ARP address family hooks

Bridge address family

The bridge address family handles Ethernet packets traversing bridge devices.

The list of supported hooks is identical to IPv4/IPv6/Inet address families above.

Netdev address family

The Netdev address family handles packets from ingress.

Hook	Description
ingress	All packets entering the system are processed by this hook. It is invoked before layer 3 protocol handlers and it can be used for early filtering and policing.

Table 3: Netdev address family hooks

Ruleset

list | flush`ruleset` [*family*]

export [**ruleset**] *format*

The **ruleset** keyword is used to identify the whole set of tables, chains, etc. currently in place in kernel. The following **ruleset** commands exist:

`list` Print the ruleset in human-readable format.

`flush` Clear the whole ruleset. Note that unlike iptables, this will remove all tables and whatever they contain, effectively leading to an empty ruleset - no packet filtering will happen anymore, so the kernel accepts any valid packet it receives.

`export` Print the ruleset in machine readable format. The mandatory *format* parameter may be either `xml` or `json`.

It is possible to limit **list** and **flush** to a specific address family only. For a list of valid family names, see `ADDRESS FAMILIES` above.

Note that contrary to what one might assume, the output generated by **export** is not parseable by **nft -f**. Instead, the output of **list** command serves well for that purpose.

Tables

add | create`table` [*family*]*table* [flags *flags*]

delete | list | flush`table` [*family*]*table*

`deletetable` [*family*] `handle` *handle*

Tables are containers for chains, sets and stateful objects. They are identified by their address family and their name. The address family must be one of `ip`, `ip6`, `inet`, `arp`, `bridge`, `netdev`. The `inet` address family is a dummy family which is used to create hybrid IPv4/IPv6 tables. The `meta` expression `nfproto` keyword can be used to test which family (IPv4 or IPv6) context the packet is being processed in. When no address family is specified, `ip` is used by default. The only difference between **add** and **create** is that the former will not return an error if the specified table already exists while **create** will return an error.

Flag	Description
dormant	table is not evaluated any more (base chains are unregistered)

Table 4: Table flags

Example 1.2 Add, change, delete a table

```
# start nft in interactive mode
nft --interactive

# create a new table.
create table inet mytable

# add a new base chain: get input packets
add chain inet mytable myin { type filter hook input priority 0; }

# add a single counter to the chain
add rule inet mytable myin counter

# disable the table temporarily -- rules are not evaluated anymore
add table inet mytable { flags dormant; }

# make table active again:
add table inet mytable
```

add Add a new table for the given family with the given name.

delete Delete the specified table.

list List all chains and rules of the specified table.

flush Flush all chains and rules of the specified table.

Chains

add | create `chain` [*family*] *table chain* [type *type* hook *hook* [device *device*] priority *priority* ; [policy *policy* ;]]

delete | list | flush `chain` [*family*] *table chain*

`deletechain` [*family*] *table* `handle` *handle*

`renamechain` [*family*] *table chain new name*

Chains are containers for rules. They exist in two kinds, base chains and regular chains. A base chain is an entry point for packets from the networking stack, a regular chain may be used as jump target and is used for better rule organization.

add Add a new chain in the specified table. When a hook and priority value are specified, the chain is created as a base chain and hooked up to the networking stack.

create Similar to the **add** command, but returns an error if the chain already exists.

delete Delete the specified chain. The chain must not contain any rules or be used as jump target.

rename Rename the specified chain.

list List all rules of the specified chain.

flush Flush all rules of the specified chain.

For base chains, **type**, **hook** and **priority** parameters are mandatory.

Apart from the special cases illustrated above (e.g. `nat` type not supporting `forward` hook or `route` type only supporting `output` hook), there are two further quirks worth noticing:

- `netdev` family supports merely a single combination, namely `filter` type and `ingress` hook. Base chains in this family also require the `device` parameter to be present since they exist per incoming interface only.
- `arp` family supports only `input` and `output` hooks, both in chains of type `filter`.

The `priority` parameter accepts a signed integer value which specifies the order in which chains with same `hook` value are traversed. The ordering is ascending, i.e. lower priority values have precedence over higher ones.

Base chains also allow to set the chain's `policy`, i.e. what happens to packets not explicitly accepted or refused in contained rules. Supported policy values are `accept` (which is the default) or `drop`.

Rules

[add | insert] `rule` [*family*] *table chain* [handle | position *handle* | `index` *index*] *statement*...

`replace rule` [*family*] *table chain* `handle` *handle statement*...

`deleterule` [*family*] *table chain* `handle` *handle*

Rules are added to `chain` in the given `table`. If the `family` is not specified, the `ip` family is used. Rules are constructed from two kinds of components according to a set of grammatical rules: expressions and statements.

Type	Families	Hooks	Description
filter	all	all	Standard chain type to use in doubt.
nat	ip, ip6	prerouting, input, output, postrouting	Chains of this type perform Network Address Translation based on conntrack entries. Only the first packet of a connection actually traverses this chain - its rules usually define details of the created conntrack entry (NAT statements for instance).
route	ip, ip6	output	If a packet has traversed a chain of this type and is about to be accepted, a new route lookup is performed if relevant parts of the IP header have changed. This allows to e.g. implement policy routing selectors in nftables.

Table 5: Supported chain types

The `add` and `insert` commands support an optional location specifier, which is either a *`handle`* of an existing rule or an *`index`* (starting at zero). Internally, rule locations are always identified by *`handle`* and the translation from *`index`* happens in userspace. This has two potential implications in case a concurrent ruleset change happens after the translation was done: The effective rule index might change if a rule was inserted or deleted before the referred one. If the referred rule was deleted, the command is rejected by the kernel just as if an invalid *`handle`* was given.

`add` Add a new rule described by the list of statements. The rule is appended to the given chain unless a `handle` is specified, in which case the rule is appended to the rule given by the *`handle`*. The alternative name `position` is deprecated and should not be used anymore.

`insert` Similar to the **add** command, but the rule is prepended to the beginning of the chain or before the rule with the given *`handle`*.

`replace` Similar to the **add** command, but the rule replaces the specified rule.

`delete` Delete the specified rule.

Example 1.3 add a rule to ip table input chain

```
nft add rule filter output ip daddr 192.168.0.0/24 accept # 'ip filter' is assumed
# same command, slightly more verbose
nft add rule ip filter output ip daddr 192.168.0.0/24 accept
```

Example 1.4 delete rule from inet table

```
# nft -a list ruleset
table inet filter {
        chain input {
                type filter hook input priority 0; policy accept;
                ct state established,related accept # handle 4
```

```
                ip saddr 10.1.1.1 tcp dport ssh accept # handle 5
    ...
# delete the rule with handle 5
# nft delete rule inet filter input handle 5
```

Sets

nftables offers two kinds of set concepts. Anonymous sets are sets that have no specific name. The set members are enclosed in curly braces, with commas to separate elements when creating the rule the set is used in. Once that rule is removed, the set is removed as well. They cannot be updated, i.e. once an anonymous set is declared it cannot be changed anymore except by removing/altering the rule that uses the anonymous set.

Example 1.5 Using anonymous sets to accept particular subnets and ports

```
        nft add rule filter input ip saddr { 10.0.0.0/8, 192.168.0.0/16 } tcp dport { 22, ↩
            443 } accept
```

Named sets are sets that need to be defined first before they can be referenced in rules. Unlike anonymous sets, elements can be added to or removed from a named set at any time. Sets are referenced from rules using an `@` prefixed to the sets name.

Example 1.6 Using named sets to accept addresses and ports

```
        nft add rule filter input ip saddr @allowed_hosts tcp dport @allowed_ports accept
```

`allowed_hostsallowed_ports`

`add set` [*`family`*]*`tableset`* { type *`type`* ; [flags *`flags`* ;] [timeout *`timeout`* ;] [gc-interval *`gc-interval`* ;] [elements = { *`element`*[,...] } ;] [size *`size`* ;] [policy *`policy`* ;] [auto-merge *`auto-merge`* ;] }

delete | list | flush `set` [*`family`*]*`tableset`*

`delete set` [*`family`*]*`table`* `handle` *`handle`*

add | delete `element` [*`family`*]*`tableset`* { *`element`*[,...] }

Sets are elements containers of an user-defined data type, they are uniquely identified by an user-defined name and attached to tables. Their behaviour can be tuned with the `flags` that can be specified at set creation time.

`add` Add a new set in the specified table. See the `Set specification` table below for more information about how to specify a sets properties.

`delete` Delete the specified set.

`list` Display the elements in the specified set.

`flush` Remove all elements from the specified set.

`add element` Comma-separated list of elements to add into the specified set.

`delete element` Comma-separated list of elements to delete from the specified set.

Maps

`add map` [*`family`*]*`tablemap`* { type *`type`* [flags *`flags`* ;] [elements = { *`element`*[,...] } ;] [size *`size`* ;] [policy *`policy`* ;] }

delete | list | flush `map` [*`family`*]*`tablemap`*

add | delete `element` [*`family`*]*`tablemap`* { elements = { *`element`*[,...] } ; }

Maps store data based on some specific key used as input, they are uniquely identified by an user-defined name and attached to tables.

Keyword	Description	Type
type	data type of set elements	string: ipv4_addr, ipv6_addr, ether_addr, inet_proto, inet_service, mark
flags	set flags	string: constant, dynamic, interval, timeout
timeout	time an element stays in the set, mandatory if set is added to from the packet path (ruleset).	string, decimal followed by unit. Units are: d, h, m, s
gc-interval	garbage collection interval, only available when timeout or flag timeout are active	string, decimal followed by unit. Units are: d, h, m, s
elements	elements contained by the set	set data type
size	maximum number of elements in the set, mandatory if set is added to from the packet path (ruleset).	unsigned integer (64 bit)
policy	set policy	string: performance [default], memory
auto-merge	automatic merge of adjacent/overlapping set elements (only for interval sets)	

Table 6: Set specifications

`add` Add a new map in the specified table.

`delete` Delete the specified map.

`list` Display the elements in the specified map.

`flush` Remove all elements from the specified map.

`add element` Comma-separated list of elements to add into the specified map.

`delete element` Comma-separated list of element keys to delete from the specified map.

Keyword	Description	Type
type	data type of map elements	string ':' string: ipv4_addr, ipv6_addr, ether_addr, inet_proto, inet_service, mark, counter, quota. Counter and quota can't be used as keys
flags	map flags	string: constant, interval
elements	elements contained by the map	map data type
size	maximum number of elements in the map	unsigned integer (64 bit)
policy	map policy	string: performance [default], memory

Table 7: Map specifications

Flowtables

add | create`flowtable` [*`family`*] *`table flowtable`* hook *`hook`* priority *`priority`* ; devices = { *`device`*[,...] } ;

delete | list`flowtable` [*`family`*]*`tableflowtable`*

Flowtables allow you to accelerate packet forwarding in software. Flowtables entries are represented through a tuple that is composed of the input interface, source and destination address, source and destination port; and layer 3/4 protocols. Each entry

also caches the destination interface and the gateway address - to update the destination link-layer address - to forward packets. The ttl and hoplimit fields are also decremented. Hence, flowtables provides an alternative path that allow packets to bypass the classic forwarding path. Flowtables reside in the ingress hook, that is located before the prerouting hook. You can select what flows you want to offload through the `flow offload` expression from the `forward` chain. Flowtables are identified by their address family and their name. The address family must be one of `ip`, `ip6`, `inet`. The `inet` address family is a dummy family which is used to create hybrid IPv4/IPv6 tables. When no address family is specified, `ip` is used by default.

`add` Add a new flowtable for the given family with the given name.

`delete` Delete the specified flowtable.

`list` List all flowtables.

Stateful objects

add | delete | list | reset `type` [*`family`*]*`tableobject`*

`delete type` [*`family`*]*`table`* `handle` *`handle`*

Stateful objects are attached to tables and are identified by a unique name. They group stateful information from rules, to reference them in rules the keywords "type name" are used e.g. "counter name".

`add` Add a new stateful object in the specified table.

`delete` Delete the specified object.

`list` Display stateful information the object holds.

`reset` List-and-reset stateful object.

Ct

`ct helper`*`helper`* `{ type` *`type`* `protocol` *`protocol`* `;` [`l3proto` *`family`* `;`] `}`

Ct helper is used to define connection tracking helpers that can then be used in combination with the "`ct helper set`" statement. type and protocol are mandatory, l3proto is derived from the table family by default, i.e. in the inet table the kernel will try to load both the IPv4 and IPv6 helper backends, if they are supported by the kernel.

Keyword	Description	Type
type	name of helper type	quoted string (e.g. "ftp")
protocol	layer 4 protocol of the helper	string (e.g. tcp)
l3proto	layer 3 protocol of the helper	address family (e.g. ip)

Table 8: conntrack helper specifications

Example 1.7 defining and assigning ftp helper

Unlike iptables, helper assignment needs to be performed after the conntrack lookup has completed, for example with the default 0 hook priority.

```
table inet myhelpers {
  ct helper ftp-standard {
     type "ftp" protocol tcp
  }
  chain prerouting {
      type filter hook prerouting priority 0;
      tcp dport 21 ct helper set "ftp-standard"
  }
}
```

Counter

`counter` [packets bytes]

Keyword	Description	Type
packets	initial count of packets	unsigned integer (64 bit)
bytes	initial count of bytes	unsigned integer (64 bit)

Table 9: Counter specifications

Quota

`quota` [over | until] [used]

Keyword	Description	Type
quota	quota limit, used as the quota name	Two arguments, unsigned integer (64 bit) and string: bytes, kbytes, mbytes. "over" and "until" go before these arguments
used	initial value of used quota	Two arguments, unsigned integer (64 bit) and string: bytes, kbytes, mbytes

Table 10: Quota specifications

Expressions

Expressions represent values, either constants like network addresses, port numbers etc. or data gathered from the packet during ruleset evaluation. Expressions can be combined using binary, logical, relational and other types of expressions to form complex or relational (match) expressions. They are also used as arguments to certain types of operations, like NAT, packet marking etc.

Each expression has a data type, which determines the size, parsing and representation of symbolic values and type compatibility with other expressions.

describe command

`describe` *`expression`*

The **describe** command shows information about the type of an expression and its data type.

Example 1.8 The **describe** command

```
$ nft describe tcp flags
payload expression, datatype tcp_flag (TCP flag) (basetype bitmask, integer), 8 bits

predefined symbolic constants:
fin                             0x01
syn                             0x02
rst                             0x04
psh                             0x08
ack                             0x10
urg                             0x20
ecn                             0x40
cwr                             0x80
```

Data types

Data types determine the size, parsing and representation of symbolic values and type compatibility of expressions. A number of global data types exist, in addition some expression types define further data types specific to the expression type. Most data types have a fixed size, some however may have a dynamic size, f.i. the string type.

Types may be derived from lower order types, f.i. the IPv4 address type is derived from the integer type, meaning an IPv4 address can also be specified as an integer value.

In certain contexts (set and map definitions) it is necessary to explicitly specify a data type. Each type has a name which is used for this.

Integer type

Name	Keyword	Size	Base type
Integer	integer	variable	-

The integer type is used for numeric values. It may be specified as decimal, hexadecimal or octal number. The integer type doesn't have a fixed size, its size is determined by the expression for which it is used.

Bitmask type

Name	Keyword	Size	Base type
Bitmask	bitmask	variable	integer

The bitmask type (**bitmask**) is used for bitmasks.

String type

Name	Keyword	Size	Base type
String	string	variable	-

The string type is used for character strings. A string begins with an alphabetic character (a-zA-Z) followed by zero or more alphanumeric characters or the characters `/`, `-`, `_` and `.`. In addition anything enclosed in double quotes (`"`) is recognized as a string.

Example 1.9 String specification

```
# Interface name
filter input iifname eth0

# Weird interface name
filter input iifname "(eth0)"
```

Link layer address type

The link layer address type is used for link layer addresses. Link layer addresses are specified as a variable amount of groups of two hexadecimal digits separated using colons (`:`).

Name	Keyword	Size	Base type
Link layer address	lladdr	variable	integer

Example 1.10 Link layer address specification

```
# Ethernet destination MAC address
filter input ether daddr 20:c9:d0:43:12:d9
```

IPv4 address type

Name	Keyword	Size	Base type
IPv4 address	ipv4_addr	32 bit	integer

The IPv4 address type is used for IPv4 addresses. Addresses are specified in either dotted decimal, dotted hexadecimal, dotted octal, decimal, hexadecimal, octal notation or as a host name. A host name will be resolved using the standard system resolver.

Example 1.11 IPv4 address specification

```
# dotted decimal notation
filter output ip daddr 127.0.0.1

# host name
filter output ip daddr localhost
```

IPv6 address type

Name	Keyword	Size	Base type
IPv6 address	ipv6_addr	128 bit	integer

The IPv6 address type is used for IPv6 addresses. Addresses are specified as a host name or as hexadecimal halfwords separated by colons. Addresses might be enclosed in square brackets ("[]") to differentiate them from port numbers.

Example 1.13 IPv6 address specification with bracket notation

```
# abbreviated loopback address
filter output ip6 daddr ::1
```

```
# without [] the port number (22) would be parsed as part of ipv6 address
ip6 nat prerouting tcp dport 2222 dnat to [1ce::d0]:22
```

Boolean type

The boolean type is a syntactical helper type in user space. It's use is in the right-hand side of a (typically implicit) relational expression to change the expression on the left-hand side into a boolean check (usually for existence).

The following keywords will automatically resolve into a boolean type with given value:

Name	Keyword	Size	Base type
Boolean	boolean	1 bit	integer

Keyword	Value
exists	1
missing	0

Example 1.14 Boolean specification

The following expressions support a boolean comparison:

```
# match if route exists
filter input fib daddr . iif oif exists

# match only non-fragmented packets in IPv6 traffic
filter input exthdr frag missing

# match if TCP timestamp option is present
filter input tcp option timestamp exists
```

ICMP Type type

The ICMP Type type is used to conveniently specify the ICMP header's type field.

The following keywords may be used when specifying the ICMP type:

Example 1.15 ICMP Type specification

```
# match ping packets
filter output icmp type { echo-request, echo-reply }
```

ICMP Code type

The ICMP Code type is used to conveniently specify the ICMP header's code field.

The following keywords may be used when specifying the ICMP code:

ICMPv6 Type type

The ICMPv6 Type type is used to conveniently specify the ICMPv6 header's type field.

The following keywords may be used when specifying the ICMPv6 type:

Example 1.16 ICMPv6 Type specification

```
# match ICMPv6 ping packets
filter output icmpv6 type { echo-request, echo-reply }
```

Expression	Behaviour
fib	Check route existence.
exthdr	Check IPv6 extension header existence.
tcp option	Check TCP option header existence.

Name	Keyword	Size	Base type
ICMP Type	icmp_type	8 bit	integer

Keyword	Value
echo-reply	0
destination-unreachable	3
source-quench	4
redirect	5
echo-request	8
router-advertisement	9
router-solicitation	10
time-exceeded	11
parameter-problem	12
timestamp-request	13
timestamp-reply	14
info-request	15
info-reply	16
address-mask-request	17
address-mask-reply	18

Name	Keyword	Size	Base type
ICMP Code	icmp_code	8 bit	integer

Keyword	Value
net-unreachable	0
host-unreachable	1
prot-unreachable	2
port-unreachable	3
net-prohibited	9
host-prohibited	10
admin-prohibited	13

Name	Keyword	Size	Base type
ICMPv6 Type	icmpv6_type	8 bit	integer

Keyword	Value
destination-unreachable	1
packet-too-big	2
time-exceeded	3
parameter-problem	4
echo-request	128
echo-reply	129
mld-listener-query	130
mld-listener-report	131
mld-listener-done	132
mld-listener-reduction	132
nd-router-solicit	133
nd-router-advert	134
nd-neighbor-solicit	135
nd-neighbor-advert	136
nd-redirect	137
router-renumbering	138
ind-neighbor-solicit	141
ind-neighbor-advert	142
mld2-listener-report	143

ICMPv6 Code type

Name	Keyword	Size	Base type
ICMPv6 Code	icmpv6_code	8 bit	integer

The ICMPv6 Code type is used to conveniently specify the ICMPv6 header's code field.

The following keywords may be used when specifying the ICMPv6 code:

Keyword	Value
no-route	0
admin-prohibited	1
addr-unreachable	3
port-unreachable	4
policy-fail	5
reject-route	6

ICMPvX Code type

The ICMPvX Code type abstraction is a set of values which overlap between ICMP and ICMPv6 Code types to be used from the inet family.

The following keywords may be used when specifying the ICMPvX code:

Conntrack types

This is an overview of types used in **ct** expression and statement:

For each of the types above, keywords are available for convenience:

Possible keywords for conntrack label type (**ct_label**) are read at runtime from `/etc/connlabel.conf`.

Name	Keyword	Size	Base type
ICMPvX Code	icmpx_code	8 bit	integer

Keyword	Value
no-route	0
port-unreachable	1
host-unreachable	2
admin-prohibited	3

Name	Keyword	Size	Base type
conntrack state	ct_state	4 byte	bitmask
conntrack direction	ct_dir	8 bit	integer
conntrack status	ct_status	4 byte	bitmask
conntrack event bits	ct_event	4 byte	bitmask
conntrack label	ct_label	128 bit	bitmask

Keyword	Value
invalid	1
established	2
related	4
new	8
untracked	64

Table 11: conntrack state (ct_state)

Keyword	Value
original	0
reply	1

Table 12: conntrack direction (ct_dir)

Keyword	Value
expected	1
seen-reply	2
assured	4
confirmed	8
snat	16
dnat	32
dying	512

Table 13: conntrack status (ct_status)

Keyword	Value
new	1
related	2
destroy	4
reply	8
assured	16
protoinfo	32
helper	64
mark	128
seqadj	256
secmark	512
label	1024

Table 14: conntrack event bits (ct_event)

Primary expressions

The lowest order expression is a primary expression, representing either a constant or a single datum from a packet's payload, meta data or a stateful module.

Meta expressions

`meta` length | nfproto | l4proto | protocol | priority

[meta] mark | iif | iifname | iiftype | oif | oifname | oiftype | skuid | skgid | nftrace | rtclassid | ibrname | obrname | pkttype | cpu | iifgroup | oifgroup | cgroup | random | secpath

A meta expression refers to meta data associated with a packet.

There are two types of meta expressions: unqualified and qualified meta expressions. Qualified meta expressions require the **meta** keyword before the meta key, unqualified meta expressions can be specified by using the meta key directly or as qualified meta expressions. Meta l4proto is useful to match a particular transport protocol that is part of either an IPv4 or IPv6 packet. It will also skip any IPv6 extension headers present in an IPv6 packet.

Example 1.17 Using meta expressions

```
# qualified meta expression
filter output meta oif eth0

# unqualified meta expression
filter output oif eth0

# packed was subject to ipsec processing
raw prerouting meta secpath exists accept
```

socket expression

`socket` transparent

Socket expression can be used to search for an existing open TCP/UDP socket and its attributes that can be associated with a packet. It looks for an established or non-zero bound listening socket (possibly with a non-local address).

Keyword	Description	Type
length	Length of the packet in bytes	integer (32 bit)
nfproto	real hook protocol family, useful only in inet table	integer (32 bit)
l4proto	layer 4 protocol, skips ipv6 extension headers	integer (8 bit)
protocol	EtherType protocol value	ether_type
priority	TC packet priority	tc_handle
mark	Packet mark	mark
iif	Input interface index	iface_index
iifname	Input interface name	ifname
iiftype	Input interface type	iface_type
oif	Output interface index	iface_index
oifname	Output interface name	ifname
oiftype	Output interface hardware type	iface_type
skuid	UID associated with originating socket	uid
skgid	GID associated with originating socket	gid
rtclassid	Routing realm	realm
ibrname	Input bridge interface name	ifname
obrname	Output bridge interface name	ifname
pkttype	packet type	pkt_type
cpu	cpu number processing the packet	integer (32 bits)
iifgroup	incoming device group	devgroup
oifgroup	outgoing device group	devgroup
cgroup	control group id	integer (32 bits)
random	pseudo-random number	integer (32 bits)
secpath	boolean	boolean (1 bit)

Table 15: Meta expression types

Type	Description
iface_index	Interface index (32 bit number). Can be specified numerically or as name of an existing interface.
ifname	Interface name (16 byte string). Does not have to exist.
iface_type	Interface type (16 bit number).
uid	User ID (32 bit number). Can be specified numerically or as user name.
gid	Group ID (32 bit number). Can be specified numerically or as group name.
realm	Routing Realm (32 bit number). Can be specified numerically or as symbolic name defined in /etc/iproute2/rt_realms.
devgroup_type	Device group (32 bit number). Can be specified numerically or as symbolic name defined in /etc/iproute2/group.
pkt_type	Packet type: Unicast (addressed to local host), Broadcast (to all), Multicast (to group).

Table 16: Meta expression specific types

Name	Description	Type
transparent	Value of the IP_TRANSPARENT socket option in the found socket. It can be 0 or 1.	boolean (1 bit)

Table 17: Available socket attributes

Example 1.18 Using socket expression

```
# Mark packets that correspond to a transparent socket
table inet x {
    chain y {
        type filter hook prerouting priority -150; policy accept;
        socket transparent 1 mark set 0x00000001 accept
    }
}
```

fib expressions

`fib` saddr | daddr mark | iif | oif oif | oifname | type

A fib expression queries the fib (forwarding information base) to obtain information such as the output interface index a particular address would use. The input is a tuple of elements that is used as input to the fib lookup functions.

Keyword	Description	Type
oif	Output interface index	integer (32 bit)
oifname	Output interface name	string
type	Address type	fib_addrtype

Table 18: fib expression specific types

Example 1.19 Using fib expressions

```
# drop packets without a reverse path
filter prerouting fib saddr . iif oif missing drop

# drop packets to address not configured on ininterface
filter prerouting fib daddr . iif type != { local, broadcast, multicast } drop

# perform lookup in a specific 'blackhole' table (0xdead, needs ip appropriate ip rule)
filter prerouting meta mark set 0xdead fib daddr . mark type vmap { blackhole : drop,  ↩
    prohibit : jump prohibited, unreachable : drop }
```

Routing expressions

`rt` classid | nexthop

A routing expression refers to routing data associated with a packet.

Example 1.20 Using routing expressions

```
# IP family independent rt expression
filter output rt classid 10
```

Keyword	Description	Type
classid	Routing realm	realm
nexthop	Routing nexthop	ipv4_addr/ipv6_addr
mtu	TCP maximum segment size of route	integer (16 bit)

Table 19: Routing expression types

Type	Description
realm	Routing Realm (32 bit number). Can be specified numerically or as symbolic name defined in /etc/iproute2/rt_realms.

Table 20: Routing expression specific types

```
# IP family dependent rt expressions
ip filter output rt nexthop 192.168.0.1
ip6 filter output rt nexthop fd00::1
inet filter output rt ip nexthop 192.168.0.1
inet filter output rt ip6 nexthop fd00::1
```

Payload expressions

Payload expressions refer to data from the packet's payload.

Ethernet header expression

`ether` [*`Ethernet header field`*]

Keyword	Description	Type
daddr	Destination MAC address	ether_addr
saddr	Source MAC address	ether_addr
type	EtherType	ether_type

Table 21: Ethernet header expression types

VLAN header expression

`vlan` [*`VLAN header field`*]

ARP header expression

`arp` [*`ARP header field`*]

IPv4 header expression

`ip` [*`IPv4 header field`*]

ICMP header expression

`icmp` [*`ICMP header field`*]

Keyword	Description	Type
id	VLAN ID (VID)	integer (12 bit)
cfi	Canonical Format Indicator	integer (1 bit)
pcp	Priority code point	integer (3 bit)
type	EtherType	ether_type

Table 22: VLAN header expression

Keyword	Description	Type
htype	ARP hardware type	integer (16 bit)
ptype	EtherType	ether_type
hlen	Hardware address len	integer (8 bit)
plen	Protocol address len	integer (8 bit)
operation	Operation	arp_op

Table 23: ARP header expression

Keyword	Description	Type
version	IP header version (4)	integer (4 bit)
hdrlength	IP header length including options	integer (4 bit) FIXME scaling
dscp	Differentiated Services Code Point	dscp
ecn	Explicit Congestion Notification	ecn
length	Total packet length	integer (16 bit)
id	IP ID	integer (16 bit)
frag-off	Fragment offset	integer (16 bit)
ttl	Time to live	integer (8 bit)
protocol	Upper layer protocol	inet_proto
checksum	IP header checksum	integer (16 bit)
saddr	Source address	ipv4_addr
daddr	Destination address	ipv4_addr

Table 24: IPv4 header expression

Keyword	Description	Type
type	ICMP type field	icmp_type
code	ICMP code field	integer (8 bit)
checksum	ICMP checksum field	integer (16 bit)
id	ID of echo request/response	integer (16 bit)
sequence	sequence number of echo request/response	integer (16 bit)
gateway	gateway of redirects	integer (32 bit)
mtu	MTU of path MTU discovery	integer (16 bit)

Table 25: ICMP header expression

IPv6 header expression

`ip6` [*IPv6 header field*]

This expression refers to the ipv6 header fields. Caution when using **ip6 nexthdr**, the value only refers to the next header, i.e. **ip6 nexthdr tcp** will only match if the ipv6 packet does not contain any extension headers. Packets that are fragmented or e.g. contain a routing extension headers will not be matched. Please use **meta l4proto** if you wish to match the real transport header and ignore any additional extension headers instead.

Keyword	Description	Type
version	IP header version (6)	integer (4 bit)
dscp	Differentiated Services Code Point	dscp
ecn	Explicit Congestion Notification	ecn
flowlabel	Flow label	integer (20 bit)
length	Payload length	integer (16 bit)
nexthdr	Nexthdr protocol	inet_proto
hoplimit	Hop limit	integer (8 bit)
saddr	Source address	ipv6_addr
daddr	Destination address	ipv6_addr

Table 26: IPv6 header expression

Example 1.21 matching if first extension header indicates a fragment

```
ip6 nexthdr ipv6-frag counter
```

ICMPv6 header expression

`icmpv6` [*ICMPv6 header field*]

Keyword	Description	Type
type	ICMPv6 type field	icmpv6_type
code	ICMPv6 code field	integer (8 bit)
checksum	ICMPv6 checksum field	integer (16 bit)
parameter-problem	pointer to problem	integer (32 bit)
packet-too-big	oversized MTU	integer (32 bit)
id	ID of echo request/response	integer (16 bit)
sequence	sequence number of echo request/response	integer (16 bit)
max-delay	maximum response delay of MLD queries	integer (16 bit)

Table 27: ICMPv6 header expression

TCP header expression

`tcp` [*TCP header field*]

UDP header expression

`udp` [*UDP header field*]

Keyword	Description	Type
sport	Source port	inet_service
dport	Destination port	inet_service
sequence	Sequence number	integer (32 bit)
ackseq	Acknowledgement number	integer (32 bit)
doff	Data offset	integer (4 bit) FIXME scaling
reserved	Reserved area	integer (4 bit)
flags	TCP flags	tcp_flag
window	Window	integer (16 bit)
checksum	Checksum	integer (16 bit)
urgptr	Urgent pointer	integer (16 bit)

Table 28: TCP header expression

Keyword	Description	Type
sport	Source port	inet_service
dport	Destination port	inet_service
length	Total packet length	integer (16 bit)
checksum	Checksum	integer (16 bit)

Table 29: UDP header expression

UDP-Lite header expression

`udplite` [*UDP-Lite header field*]

Keyword	Description	Type
sport	Source port	inet_service
dport	Destination port	inet_service
checksum	Checksum	integer (16 bit)

Table 30: UDP-Lite header expression

SCTP header expression

`sctp` [*SCTP header field*]

DCCP header expression

`dccp` [*DCCP header field*]

Authentication header expression

`ah` [*AH header field*]

Encrypted security payload header expression

`esp` [*ESP header field*]

IPComp header expression

`comp` [*IPComp header field*]

Keyword	Description	Type
sport	Source port	inet_service
dport	Destination port	inet_service
vtag	Verification Tag	integer (32 bit)
checksum	Checksum	integer (32 bit)

Table 31: SCTP header expression

Keyword	Description	Type
sport	Source port	inet_service
dport	Destination port	inet_service

Table 32: DCCP header expression

Keyword	Description	Type
nexthdr	Next header protocol	inet_proto
hdrlength	AH Header length	integer (8 bit)
reserved	Reserved area	integer (16 bit)
spi	Security Parameter Index	integer (32 bit)
sequence	Sequence number	integer (32 bit)

Table 33: AH header expression

Keyword	Description	Type
spi	Security Parameter Index	integer (32 bit)
sequence	Sequence number	integer (32 bit)

Table 34: ESP header expression

Keyword	Description	Type
nexthdr	Next header protocol	inet_proto
flags	Flags	bitmask
cpi	Compression Parameter Index	integer (16 bit)

Table 35: IPComp header expression

Raw payload expression

`@ [base,offset,length]`

The raw payload expression instructs to load *length*bits starting at *offset*bits. Bit 0 refers to the very first bit -- in the C programming language, this corresponds to the topmost bit, i.e. 0x80 in case of an octet. They are useful to match headers that do not have a human-readable template expression yet. Note that nft will not add dependencies for Raw payload expressions. If you e.g. want to match protocol fields of a transport header with protocol number 5, you need to manually exclude packets that have a different transport header, for instance my using `meta l4proto 5` before the raw expression.

Base	Description
ll	Link layer, for example the Ethernet header
nh	Network header, for example IPv4 or IPv6
th	Transport Header, for example TCP

Table 36: Supported payload protocol bases

Example 1.22 Matching destination port of both UDP and TCP

```
inet filter input meta l4proto {tcp, udp} @th,16,16 { dns, http }
```

Example 1.23 Rewrite arp packet target hardware address if target protocol address matches a given address

```
input meta iifname enp2s0 arp ptype 0x0800 arp htype 1 arp hlen 6 arp plen 4 @nh,192,32 0 ↩
    xc0a88f10 @nh,144,48 set 0x112233445566 accept
```

Extension header expressions

Extension header expressions refer to data from variable-sized protocol headers, such as IPv6 extension headers and TCP options.

nftables currently supports matching (finding) a given ipv6 extension header or TCP option.

`hbh` nexthdr | hdrlength

`frag` nexthdr | frag-off | more-fragments | id

`rt` nexthdr | hdrlength | type | seg-left

`dst` nexthdr | hdrlength

`mh` nexthdr | hdrlength | checksum | type

`srh` flags | tag | sid | seg-left

`tcp option` eol | noop | maxseg | window | sack-permitted | sack | sack0 | sack1 | sack2 | sack3 | timestamp [*tcp_option_field*]

The following syntaxes are valid only in a relational expression with boolean type on right-hand side for checking header existence only:

`exthdr` hbh | frag | rt | dst | mh

`tcp option` eol | noop | maxseg | window | sack-permitted | sack | sack0 | sack1 | sack2 | sack3 | timestamp

Example 1.24 finding TCP options

```
filter input tcp option sack-permitted kind 1 counter
```

Keyword	Description
hbh	Hop by Hop
rt	Routing Header
frag	Fragmentation header
dst	dst options
mh	Mobility Header
srh	Segment Routing Header

Table 37: IPv6 extension headers

Keyword	Description	TCP option fields
eol	End of option list	kind
noop	1 Byte TCP No-op options	kind
maxseg	TCP Maximum Segment Size	kind, length, size
window	TCP Window Scaling	kind, length, count
sack-permitted	TCP SACK permitted	kind, length
sack	TCP Selective Acknowledgement (alias of block 0)	kind, length, left, right
sack0	TCP Selective Acknowledgement (block 0)	kind, length, left, right
sack1	TCP Selective Acknowledgement (block 1)	kind, length, left, right
sack2	TCP Selective Acknowledgement (block 2)	kind, length, left, right
sack3	TCP Selective Acknowledgement (block 3)	kind, length, left, right
timestamp	TCP Timestamps	kind, length, tsval, tsecr

Table 38: TCP Options

Example 1.25 matching IPv6 exthdr

```
ip6 filter input frag more-fragments 1 counter
```

Conntrack expressions

Conntrack expressions refer to meta data of the connection tracking entry associated with a packet.

There are three types of conntrack expressions. Some conntrack expressions require the flow direction before the conntrack key, others must be used directly because they are direction agnostic. The **packets**, **bytes** and **avgpkt** keywords can be used with or without a direction. If the direction is omitted, the sum of the original and the reply direction is returned. The same is true for the **zone**, if a direction is given, the zone is only matched if the zone id is tied to the given direction.

`ct` state | direction | status | mark | expiration | helper | label | l3proto | protocol | bytes | packets | avgpkt | zone

`ct` original | reply l3proto | protocol | proto-src | proto-dst | bytes | packets | avgpkt | zone

`ct` original | reply ip | ip6 saddr | daddr

Keyword	Description	Type
state	State of the connection	ct_state
direction	Direction of the packet relative to the connection	ct_dir
status	Status of the connection	ct_status
mark	Connection mark	mark
expiration	Connection expiration time	time
helper	Helper associated with the connection	string
label	Connection tracking label bit or symbolic name defined in connlabel.conf in the nftables include path	ct_label
l3proto	Layer 3 protocol of the connection	nf_proto
saddr	Source address of the connection for the given direction	ipv4_addr/ipv6_addr
daddr	Destination address of the connection for the given direction	ipv4_addr/ipv6_addr
protocol	Layer 4 protocol of the connection for the given direction	inet_proto
proto-src	Layer 4 protocol source for the given direction	integer (16 bit)
proto-dst	Layer 4 protocol destination for the given direction	integer (16 bit)
packets	packet count seen in the given direction or sum of original and reply	integer (64 bit)
bytes	byte count seen, see description for **packets** keyword	integer (64 bit)
avgpkt	average bytes per packet, see description for **packets** keyword	integer (64 bit)
zone	conntrack zone	integer (16 bit)

Table 39: Conntrack expressions

A description of conntrack-specific types listed above can be found sub-section `CONNTRACK TYPES` above.

Statements

Statements represent actions to be performed. They can alter control flow (return, jump to a different chain, accept or drop the packet) or can perform actions, such as logging, rejecting a packet, etc.

Statements exist in two kinds. Terminal statements unconditionally terminate evaluation of the current rule, non-terminal statements either only conditionally or never terminate evaluation of the current rule, in other words, they are passive from the ruleset evaluation perspective. There can be an arbitrary amount of non-terminal statements in a rule, but only a single terminal statement as the final statement.

Verdict statement

The verdict statement alters control flow in the ruleset and issues policy decisions for packets.

accept | drop | queue | continue | return

jump | goto *chain*

accept Terminate ruleset evaluation and accept the packet.

drop Terminate ruleset evaluation and drop the packet.

queue Terminate ruleset evaluation and queue the packet to userspace.

continue Continue ruleset evaluation with the next rule. FIXME

return Return from the current chain and continue evaluation at the next rule in the last chain. If issued in a base chain, it is equivalent to **accept**.

jump *chain* Continue evaluation at the first rule in *chain*. The current position in the ruleset is pushed to a call stack and evaluation will continue there when the new chain is entirely evaluated of a **return** verdict is issued.

goto *chain* Similar to **jump**, but the current position is not pushed to the call stack, meaning that after the new chain evaluation will continue at the last chain instead of the one containing the goto statement.

Example 1.26 Verdict statements

```
# process packets from eth0 and the internal network in from_lan
# chain, drop all packets from eth0 with different source addresses.

filter input iif eth0 ip saddr 192.168.0.0/24 jump from_lan
filter input iif eth0 drop
```

Payload statement

The payload statement alters packet content. It can be used for example to set ip DSCP (differv) header field or ipv6 flow labels.

Example 1.27 route some packets instead of bridging

```
# redirect tcp:http from 192.160.0.0/16 to local machine for routing instead of bridging
# assumes 00:11:22:33:44:55 is local MAC address.
bridge input meta iif eth0 ip saddr 192.168.0.0/16 tcp dport 80 meta pkttype set unicast  ↩
    ether daddr set 00:11:22:33:44:55
```

Example 1.28 Set IPv4 DSCP header field

```
ip forward ip dscp set 42
```

Extension header statement

The extension header statement alters packet content in variable-sized headers. This can currently be used to alter the TCP Maximum segment size of packets, similar to TCPMSS.

Example 1.29 change tcp mss

```
tcp flags syn tcp option maxseg size set 1360
# set a size based on route information:
tcp flags syn tcp option maxseg size set rt mtu
```

Log statement

`log` [prefix *`quoted_string`*] [level *`syslog-level`*] [flags *`log-flags`*]

`log` [group *`nflog_group`*] [prefix *`quoted_string`*] [queue-threshold *`value`*] [snaplen *`size`*]

The log statement enables logging of matching packets. When this statement is used from a rule, the Linux kernel will print some information on all matching packets, such as header fields, via the kernel log (where it can be read with dmesg(1) or read in the syslog). If the group number is specified, the Linux kernel will pass the packet to nfnetlink_log which will multicast the packet through a netlink socket to the specified multicast group. One or more userspace processes may subscribe to the group to receive the packets, see libnetfilter_queue documentation for details. This is a non-terminating statement, so the rule evaluation continues after the packet is logged.

Keyword	Description	Type
prefix	Log message prefix	quoted string
level	Syslog level of logging	string: emerg, alert, crit, err, warn [default], notice, info, debug
group	NFLOG group to send messages to	unsigned integer (16 bit)
snaplen	Length of packet payload to include in netlink message	unsigned integer (32 bit)
queue-threshold	Number of packets to queue inside the kernel before sending them to userspace	unsigned integer (32 bit)

Table 40: log statement options

Flag	Description
tcp sequence	Log TCP sequence numbers.
tcp options	Log options from the TCP packet header.
ip options	Log options from the IP/IPv6 packet header.
skuid	Log the userid of the process which generated the packet.
ether	Decode MAC addresses and protocol.
all	Enable all log flags listed above.

Table 41: log-flags

Example 1.30 Using log statement

```
# log the UID which generated the packet and ip options
ip filter output log flags skuid flags ip options

# log the tcp sequence numbers and tcp options from the TCP packet
ip filter output log flags tcp sequence,options

# enable all supported log flags
```

```
ip6 filter output log flags all
```

Reject statement

`reject` [[with] icmp | icmpv6 | icmpx [type] icmp_code | icmpv6_code | icmpx_code]

`reject [ with tcp reset ]`

A reject statement is used to send back an error packet in response to the matched packet otherwise it is equivalent to drop so it is a terminating statement, ending rule traversal. This statement is only valid in the input, forward and output chains, and user-defined chains which are only called from those chains.

The different ICMP reject variants are meant for use in different table families:

Variant	Family	Type
icmp	ip	icmp_code
icmpv6	ip6	icmpv6_code
icmpx	inet	icmpx_code

For a description of the different types and a list of supported keywords refer to `DATA TYPES` section above. The common default reject value is **port-unreachable**.

Note that in bridge family, reject statement is only allowed in base chains which hook into `input` or `prerouting`.

Counter statement

A counter statement sets the hit count of packets along with the number of bytes.

`counter` [`packets`*`number`*`bytes`*`number`*]

Conntrack statement

The conntrack statement can be used to set the conntrack mark and conntrack labels.

`ct` mark | event | label | zone [set]*`value`*

The ct statement sets meta data associated with a connection. The zone id has to be assigned before a conntrack lookup takes place, i.e. this has to be done in prerouting and possibly output (if locally generated packets need to be placed in a distinct zone), with a hook priority of -300.

Keyword	Description	Value
event	conntrack event bits	bitmask, integer (32 bit)
helper	name of ct helper object to assign to the connection	quoted string
mark	Connection tracking mark	mark
label	Connection tracking label	label
zone	conntrack zone	integer (16 bit)

Table 42: Conntrack statement types

Example 1.31 save packet nfmark in conntrack

```
ct mark set meta mark
```

Example 1.32 set zone mapped via interface

```
table inet raw {
  chain prerouting {
      type filter hook prerouting priority -300;
      ct zone set iif map { "eth1" : 1, "veth1" : 2 }
  }
  chain output {
      type filter hook output priority -300;
      ct zone set oif map { "eth1" : 1, "veth1" : 2 }
  }
}
```

Example 1.33 restrict events reported by ctnetlink

```
ct event set new,related,destroy
```

Meta statement

A meta statement sets the value of a meta expression. The existing meta fields are: priority, mark, pkttype, nftrace.

`meta` mark | priority | pkttype | nftrace [set]*`value`*

A meta statement sets meta data associated with a packet.

Keyword	Description	Value
priority	TC packet priority	tc_handle
mark	Packet mark	mark
pkttype	packet type	pkt_type
nftrace	ruleset packet tracing on/off. Use **monitor trace** command to watch traces	0, 1

Table 43: Meta statement types

Limit statement

`limit` [rate] [over]*`packet_number`* [/] second | minute | hour | day [burst *`packet_number`* packets]

`limit` [rate] [over]*`byte_number`* bytes | kbytes | mbytes [/] second | minute | hour | day | week [burst *`byte_number`* bytes]

A limit statement matches at a limited rate using a token bucket filter. A rule using this statement will match until this limit is reached. It can be used in combination with the log statement to give limited logging. The **over** keyword, that is optional, makes it match over the specified rate.

Value	Description	Type
packet_number	Number of packets	unsigned integer (32 bit)
byte_number	Number of bytes	unsigned integer (32 bit)

Table 44: limit statement values

NAT statements

`snat` [to *`address`* [:port]] [persistent, random, fully-random]

`snat` [to *`address`* - *`address`* [:*`port`* - *`port`*]] [persistent, random, fully-random]

`dnat` [to *`address`* [:*`port`*]] [persistent, random, fully-random]

`dnat` [to *`address`* [:*`port`* - *`port`*]] [persistent, random, fully-random]

`masquerade` [to [:*`port`*]] [persistent, random, fully-random]

`masquerade` [to [:*`port`* - *`port`*]] [persistent, random, fully-random]

`redirect` [to [:*`port`*]] [persistent, random, fully-random]

`redirect` [to [:*`port`* - *`port`*]] [persistent, random, fully-random]

The nat statements are only valid from nat chain types.

The **snat** and **masquerade** statements specify that the source address of the packet should be modified. While **snat** is only valid in the postrouting and input chains, **masquerade** makes sense only in postrouting. The **dnat** and **redirect** statements are only valid in the prerouting and output chains, they specify that the destination address of the packet should be modified. You can use non-base chains which are called from base chains of nat chain type too. All future packets in this connection will also be mangled, and rules should cease being examined.

The **masquerade** statement is a special form of **snat** which always uses the outgoing interface's IP address to translate to. It is particularly useful on gateways with dynamic (public) IP addresses.

The **redirect** statement is a special form of **dnat** which always translates the destination address to the local host's one. It comes in handy if one only wants to alter the destination port of incoming traffic on different interfaces.

Note that all nat statements require both prerouting and postrouting base chains to be present since otherwise packets on the return path won't be seen by netfilter and therefore no reverse translation will take place.

Expression	Description	Type
address	Specifies that the source/destination address of the packet should be modified. You may specify a mapping to relate a list of tuples composed of arbitrary expression key with address value.	ipv4_addr, ipv6_addr, e.g. abcd::1234, or you can use a mapping, e.g. meta mark map { 10 : 192.168.1.2, 20 : 192.168.1.3 }
port	Specifies that the source/destination address of the packet should be modified.	port number (16 bits)

Table 45: NAT statement values

Flag	Description
persistent	Gives a client the same source-/destination-address for each connection.
random	If used then port mapping will be randomized using a random seeded MD5 hash mix using source and destination address and destination port.
fully-random	If used then port mapping is generated based on a 32-bit pseudo-random algorithm.

Table 46: NAT statement flags

Example 1.34 Using NAT statements

```
# create a suitable table/chain setup for all further examples
add table nat
add chain nat prerouting { type nat hook prerouting priority 0; }
add chain nat postrouting { type nat hook postrouting priority 100; }

# translate source addresses of all packets leaving via eth0 to address 1.2.3.4
add rule nat postrouting oif eth0 snat to 1.2.3.4

# redirect all traffic entering via eth0 to destination address 192.168.1.120
add rule nat prerouting iif eth0 dnat to 192.168.1.120

# translate source addresses of all packets leaving via eth0 to whatever
# locally generated packets would use as source to reach the same destination
add rule nat postrouting oif eth0 masquerade

# redirect incoming TCP traffic for port 22 to port 2222
add rule nat prerouting tcp dport 22 redirect to :2222
```

Flow offload statement

A flow offload statement allows us to select what flows you want to accelerate forwarding through layer 3 network stack bypass. You have to specify the flowtable name where you want to offload this flow.

`flow offload @flowtable`

Queue statement

This statement passes the packet to userspace using the nfnetlink_queue handler. The packet is put into the queue identified by its 16-bit queue number. Userspace can inspect and modify the packet if desired. Userspace must then drop or re-inject the packet into the kernel. See libnetfilter_queue documentation for details.

`queue` [num *`queue_number`*] [bypass]

`queue` [num *`queue_number_from`* - *`queue_number_to`*] [bypass,fanout]

Value	Description	Type
queue_number	Sets queue number, default is 0.	unsigned integer (16 bit)
queue_number_from	Sets initial queue in the range, if fanout is used.	unsigned integer (16 bit)
queue_number_to	Sets closing queue in the range, if fanout is used.	unsigned integer (16 bit)

Table 47: queue statement values

Flag	Description
bypass	Let packets go through if userspace application cannot back off. Before using this flag, read libnetfilter_queue documentation for performance tuning recommendations.
fanout	Distribute packets between several queues.

Table 48: queue statement flags

Dup statement

The dup statement is used to duplicate a packet and send the copy to a different destination.

`dup` [to *`device`*]

`dup` [to *`address`* [device]*`device`*]

Expression	Description	Type
address	Specifies that the copy of the packet should be sent to a new gateway.	ipv4_addr, ipv6_addr, e.g. abcd::1234, or you can use a mapping, e.g. ip saddr map { 192.168.1.2 : 10.1.1.1 }
device	Specifies that the copy should be transmitted via device.	string

Table 49: Dup statement values

Example 1.35 Using the dup statement

```
# send to machine with ip address 10.2.3.4 on eth0
ip filter forward dup to 10.2.3.4 device "eth0"

# copy raw frame to another interface
netdetv ingress dup to "eth0"
dup to "eth0"

# combine with map dst addr to gateways
dup to ip daddr map { 192.168.7.1 : "eth0", 192.168.7.2 : "eth1" }
```

Fwd statement

The fwd statement is used to redirect a raw packet to another interface. It is only available in the netdev family ingress hook. It is similar to the dup statement except that no copy is made.

`fwd` [to *`device`*]

Set statement

The set statement is used to dynamically add or update elements in a set from the packet path. The set `setname` must already exist in the given table and must have been created with the `dynamic` flag. Furthermore, these sets must specify both a maximum set size (to prevent memory exhaustion) and a timeout (so that number of entries in set will not grow indefinitely). The set statement can be used to e.g. create dynamic blacklists.

add | update*`@setname{ expression`* [timeout *`timeout`*] [comment *`string`*]*`}`*

Example 1.36 Example for simple blacklist

```
    # declare a set, bound to table "filter", in family "ip".  Timeout and size are ↩
        mandatory because we will add elements from packet path.
    nft add set ip filter blackhole "{ type ipv4_addr; flags timeout; size 65536; }"

    # whitelist internal interface.
    nft add rule ip filter input meta iifname "internal" accept

    # drop packets coming from blacklisted ip addresses.
    nft add rule ip filter input ip saddr @blackhole counter drop
```

```
    # add source ip addresses to the blacklist if more than 10 tcp connection requests  ↩
        occurred per second and ip address.
    # entries will timeout after one minute, after which they might be re-added if limit  ↩
        condition persists.
    nft add rule ip filter input tcp flags syn tcp dport ssh meter flood size 128000 { ip  ↩
        saddr timeout 10s limit rate over 10/second} add @blackhole { ip saddr timeout 1m }  ↩
        drop

    # inspect state of the rate limit meter:
    nft list meter ip filter flood

    # inspect content of blackhole:
    nft list set ip filter blackhole

    # manually add two addresses to the set:
    nft add element filter blackhole { 10.2.3.4, 10.23.1.42 }
```

Additional commands

These are some additional commands included in nft.

monitor

The monitor command allows you to listen to Netlink events produced by the nf_tables subsystem, related to creation and deletion of objects. When they occur, nft will print to stdout the monitored events in either XML, JSON or native nft format.

To filter events related to a concrete object, use one of the keywords 'tables', 'chains', 'sets', 'rules', 'elements', 'ruleset'.

To filter events related to a concrete action, use keyword 'new' or 'destroy'.

Hit ^C to finish the monitor operation.

Example 1.37 Listen to all events, report in native nft format

```
% nft monitor
```

Example 1.38 Listen to added tables, report in XML format

```
% nft monitor new tables xml
```

Example 1.39 Listen to deleted rules, report in JSON format

```
% nft monitor destroy rules json
```

Example 1.40 Listen to both new and destroyed chains, in native nft format

```
% nft monitor chains
```

Example 1.41 Listen to ruleset events such as table, chain, rule, set, counters and quotas, in native nft format

```
% nft monitor ruleset
```

Error reporting

When an error is detected, nft shows the line(s) containing the error, the position of the erroneous parts in the input stream and marks up the erroneous parts using carets (^). If the error results from the combination of two expressions or statements, the part imposing the constraints which are violated is marked using tildes (~).

For errors returned by the kernel, nft can't detect which parts of the input caused the error and the entire command is marked.

Example 1.42 Error caused by single incorrect expression

```
<cmdline>:1:19-22: Error: Interface does not exist
filter output oif eth0
                  ^^^^
```

Example 1.43 Error caused by invalid combination of two expressions

```
<cmdline>:1:28-36: Error: Right hand side of relational expression (==) must be constant
filter output tcp dport == tcp dport
                        ~~ ^^^^^^^^^
```

Example 1.44 Error returned by the kernel

```
<cmdline>:0:0-23: Error: Could not process rule: Operation not permitted
filter output oif wlan0
^^^^^^^^^^^^^^^^^^^^^^^
```

Exit status

On success, nft exits with a status of 0. Unspecified errors cause it to exit with a status of 1, memory allocation errors with a status of 2, unable to open Netlink socket with 3.

See Also

iptables(8), ip6tables(8), arptables(8), ebtables(8), ip(8), tc(8)

There is an official wiki at: https://wiki.nftables.org

Authors

nftables was written by Patrick McHardy and Pablo Neira Ayuso, among many other contributors from the Netfilter community.

Copyright