

---

## 1 nft

nft — Administration tool for packet filtering and classification

### Synopsis

```
nft [-n | --numeric] [[-I | --includepath] directory] [-f | --file] filename [-i | --interactive] | cmd
nft [-h | --help] [-v | --version]
```

### Description

nft is used to set up, maintain and inspect packet filtering and classification rules in the Linux kernel.

### Options

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

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

**-v, --version** Show version.

**-n, --numeric** Numeric output: Addresses and other information that might need network traffic to resolve to symbolic names are shown numerically (default behaviour). When used twice, internet services are translated. When used twice, internet services and UIDs/GIDs are also shown numerically. When used three times, protocol numbers are also shown numerically.

**-N** Translate IP addresses to DNS names.

**-a, --handle** Show rule handles in output.

**-I, --includepath *directory*** Add the directory *directory* to the list of directories to be searched for included files.

**-f, --file *filename*** Read input from *filename*.

**-i, --interactive** Read input from an interactive readline CLI.

### 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.

---

## 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 packets vi

**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.

**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

**Tables**

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

Tables are containers for chains and sets. 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. When no address family is specified, `ip` is used by default.

**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**

```
addchain [family] table chain hook priority policy device
```

```
add | create | delete | list | flushchain [family] table chain
```

```
renamechain [family] table chain newname
```

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.

## Rules

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

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

Rules are constructed from two kinds of components according to a set of grammatical rules: expressions and statements.

**add** Add a new rule described by the list of statements. The rule is appended to the given chain unless a position is specified, in which case the rule is appended to the rule given by the position.

**insert** Similar to the **add** command, but the rule is prepended to the beginning of the chain or before the rule at the given position.

**delete** Delete the specified rule.

## 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 1.2 The describe command

---

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

pre-defined 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.

---

---

Name	Keyword	Size	Base type
Integer	integer	variable	-

### Integer type

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 to 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 1.3 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 (:).

---

#### Example 1.4 Link layer address specification

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

---

### IPv4 address type

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.

---

Name	Keyword	Size	Base type
Link layer address	lladdr	variable	integer

Name	Keyword	Size	Base type
IPv4 address	ipv4_addr	32 bit	integer

---

#### Example 1.5 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. FIXME

---

#### Example 1.6 IPv6 address specification

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

---

### 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 | ibriport | obriport | pkttype | cpu | iifgroup | oifgroup | cgroup

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.

---

#### Example 1.7 Using meta expressions

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

# unqualified meta expression
filter output oif eth0
```

---

Keyword	Description	Type
length	Length of the packet in bytes	integer (32 bit)
protocol	Ethertype protocol value	ether_type
priority	TC packet priority	integer (32 bit)
mark	Packet mark	packetmark
iif	Input interface index	iface_index
iifname	Input interface name	string
iiftype	Input interface type	iface_type
oif	Output interface index	iface_index
oifname	Output interface name	string
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
ibriport	Input bridge interface name	string
obriport	Output bridge interface name	string
pkttype	packet type	pkt_type
cpu	cpu number processing the packet	integer (32 bits)
iifgroup	incoming device group	devgroup_type
oifgroup	outgoing device group	devgroup_type
cgroup	control group id	integer (32 bits)

Table 4: 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 5: Meta expression specific types

---

## 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 6: Ethernet header expression types

### VLAN header expression

`vlan[VLAN header field]`

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

Table 7: VLAN header expression

### ARP header expression

`arp[ARP header field]`

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

Table 8: ARP header expression

### IPv4 header expression

`ip[IPv4 header field]`

### IPv6 header expression

`ip6[IPv6 header field]`

### TCP header expression

`tcp[TCP header field]`

---



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	integer (6 bit)
ecn	Explicit Congestion Notification	integer (2 bit)
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 9: IPv4 header expression

Keyword	Description	Type
version	IP header version (6)	integer (4 bit)
priority		
dscp	Differentiated Services Code Point	integer (6 bit)
ecn	Explicit Congestion Notification	integer (2 bit)
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 10: IPv6 header expression

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_flags
window	Window	integer (16 bit)
checksum	Checksum	integer (16 bit)
urgptr	Urgent pointer	integer (16 bit)

Table 11: TCP header expression

### UDP header expression

`udp` [*UDP header field*]

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 12: 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
cscov	Checksum coverage	integer (16 bit)
checksum	Checksum	integer (16 bit)

Table 13: UDP-Lite header expression

### SCTP header expression

`sctp` [*SCTP 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 14: SCTP header expression

### 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

Table 15: DCCP header expression

Keyword	Description	Type
nexthdr	Next header protocol	inet_service
hdrlength	AH Header length	integer (8 bit)
reserved	Reserved area	integer (4 bit)
spi	Security Parameter Index	integer (32 bit)
sequence	Sequence number	integer (32 bit)

Table 16: AH header expression

## bla

### IPv6 extension header expressions

IPv6 extension header expressions refer to data from an IPv6 packet's extension headers.

### 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 and bytes** 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.

`ct state | direction | status | mark | expiration | helper | label | bytes | packets original | reply | l3proto | protocol | saddr | daddr | proto-src | proto-dst | bytes | packets`

## 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`

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

Table 17: ESP header expression

---

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

Table 18: IPComp header expression

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	packetmark
expiration	Connection expiration time	time
helper	Helper associated with the connection	string
label	Connection tracking label	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	bytecount seen, see description for <b>packets</b> keyword	integer (64 bit)

Table 19: Conntrack expressions

**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 1.8 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
```

---

**Log statement**

**Reject statement**

**Counter statement**

**Meta statement**

**Limit statement**

**NAT statement**

**Queue statement**

### Additional commands

These are some additional commands included in nft.

#### export

Export your current ruleset in XML or JSON format to stdout.

Examples:

```
% nft export xml
[...]
% nft export json
[...]
```

---

## 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'`.

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

Hit `^C` to finish the monitor operation.

---

### Example 1.9 Listen to all events, report in native nft format

---

```
% nft monitor
```

---



---

### Example 1.10 Listen to added tables, report in XML format

---

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

---



---

### Example 1.11 Listen to deleted rules, report in JSON format

---

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

---



---

### Example 1.12 Listen to both new and destroyed chains, in native nft format

---

```
% nft monitor chains
```

---

## 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 carrets (`^`). 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 1.13 Error caused by single incorrect expression

---

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

---



---

### Example 1.14 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 1.15 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: <http://wiki.nftables.org>

## Authors

nftables was written by Patrick McHardy.

## Copyright

Copyright © 2008-2014 Patrick McHardy [kaber@trash.net](mailto:kaber@trash.net)

nftables is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License version 2 as published by the Free Software Foundation.

This documentation is licenced under the terms of the Creative Commons Attribution-ShareAlike 4.0 license, [CC BY-SA 4.0](#).

---