NAME

u32 – universal 32bit traffic control filter

SYNOPSIS

tc filter ... [**handle** *HANDLE*] **u32** *OPTION_LIST* [**offset** *OFFSET*] [**hashkey** *HASHKEY*] [**classid** *CLASSID*] [**divisor** *uint_value*] [**order** *u32_value*] [**ht** *HANDLE*] [**sample** *SELECTOR* [**divisor** *uint_value*]] [**link** *HANDLE*] [**indev** *ifname*] [**skip_hw** | **skip_sw**] [**help**]

HANDLE := { *u12_hex_htid***:**[*u8_hex_hash***:**[*u12_hex_nodeid*] | **0x***u32_hex_value* }

OPTION_LIST := [*OPTION_LIST*] *OPTION*

HASHKEY := [**mask** *u32_hex_value*] [**at** *4*int_value*]

CLASSID := { **root** | **none** | [*u16_major*]**:***u16_minor* | *u32_hex_value* }

OFFSET := [**plus** *int_value*] [**at** *2*int_value*] [**mask** *u16_hex_value*] [**shift** *int_value*] [**eat**]

OPTION := { **match** *SELECTOR* | **action** *ACTION* }

SELECTOR := { **u32** *VAL_MASK_32* | **u16** *VAL_MASK_16* | **u8** *VAL_MASK_8* | **ip** *IP* | **ip6** *IP6* | { **tcp** | **udp** } *TCPUDP* | **icmp** *ICMP* | **mark** *VAL_MASK_32* | **ether** *ETHER* }

IP := { { **src** | **dst** } { **default** | **any** | **all** | *ip_address* [**/** { *prefixlen* | *netmask* }] } *AT* | { **dsfield** | **ihl** | **protocol** | **precedence** | **icmp_type** | **icmp_code** } *VAL_MASK_8* | { **sport** | **dport** } *VAL_MASK_16* | **nofrag** | **firstfrag** | **df** | **mf** }

IP6 := { { **src** | **dst** } { **default** | **any** | **all** | *ip6_address* [*/prefixlen*] } *AT* | **priority** *VAL_MASK_8* | { **protocol** | **icmp_type** | **icmp_code** } *VAL_MASK_8* | **flowlabel** *VAL_MASK_32* | { **sport** | **dport** } *VAL_MASK_16* }

TCPUDP := { **src** | **dst** } *VAL_MASK_16*

ICMP := { **type** *VAL_MASK_8* | **code** *VAL_MASK_8* }

ETHER := { **src** | **dst** } *ether_address AT*

VAL_MASK_32 := *u32_value u32_hex_mask* [*AT*]

VAL_MASK_16 := *u16_value u16_hex_mask* [*AT*]

VAL_MASK_8 := *u8_value u8_hex_mask* [*AT*]

AT := [**at** [**nexthdr+**] *int_value*]

DESCRIPTION

The Universal/Ugly 32bit filter allows to match arbitrary bitfields in the packet. Due to breaking everything down to values, masks and offsets, It is equally powerful and hard to use. Luckily many abstracting directives are present which allow defining rules on a higher level and therefore free the user from having to fiddle with bits and masks in many cases.

There are two general modes of invocation: The first mode creates a new filter to delegate packets to different destinations. Apart from the obvious ones, namely classifying the packet by specifying a *CLASSID* or calling an **action**, one may **link** one filter to another one (or even a list of them), effectively organizing filters into a tree-like hierarchy.

Typically filter delegation is done by means of a hash table, which leads to the second mode of invocation: it merely serves to set up these hash tables. Filters can select a hash table and provide a key selector from which a hash is to be computed and used as key to lookup the table's bucket which contains filters for further processing. This is useful if a high number of filters is in use, as the overhead of performing the hash operation and table lookup becomes negligible in that case. Using hashtables with **u32** basically involves the following pattern:

(1) Creating a new hash table, specifying it's size using the **divisor** parameter and ideally a handle by which the table can be identified. If the latter is not given, the kernel chooses one on it's own, which has to be guessed later.

(2) Creating filters which link to the created table in *(1)* using the **link** parameter and defining the packet data which the kernel will use to calculate the **hashkey**.

(3) Adding filters to buckets in the hash table from *(1).* In order to avoid having to know how exactly the kernel creates the hash key, there is the **sample** parameter, which gives sample data to hash and thereby define the table bucket the filter should be added to.

In fact, even if not explicitly requested **u32** creates a hash table for every **priority** a filter is being added with. The table's size is 1 though, so it is in fact merely a linked list.

VALUES

Options and selectors require values to be specified in a specific format, which is often non-intuitive. Therefore the terminals in *SYNOPSIS* have been given descriptive names to indicate the required format and/or maximum allowed numeric value: Prefixes *u32*, *u16* and *u8* indicate four, two and single byte unsigned values. E.g. *u16* indicates a two byte-sized value in range between 0 and 65535 (0xFFFF) inclusive. A prefix of *int* indicates a four byte signed value. A middle part of *_hex_* indicates that the value is parsed in hexadecimal format. Otherwise, the value's base is automatically detected, i.e. values prefixed with *0x* are considered hexadecimal, a leading *0* indicates octal format and decimal format otherwise. There are some values with special formatting as well: *ip_address* and *netmask* are in dotted-quad formatting as usual for IPv4 addresses. An *ip6_address* is specified in common, colon-separated hexadecimal format. Finally, *prefixlen* is an unsigned, decimal integer value in range from 0 to the address width in bits (32 for IPv4 and 128 for IPv6).

Sometimes values need to be dividable by a certain number. In that case a name of the form *N*val* was chosen, indicating that *val* must be dividable by *N*. Or the other way around: the resulting value must be a multiple of *N*.

OPTIONS

U32 recognizes the following options:

handle *HANDLE*

The handle is used to reference a filter and therefore must be unique. It consists of a hash table identifier **htid** and optional **hash** (which identifies the hash table's bucket) and **nodeid**. All these values are parsed as unsigned, hexadecimal numbers with length 12bits (**htid** and **nodeid**) or 8bits (**hash**). Alternatively one may specify a single, 32bit long hex number which contains the three fields bits in concatenated form. Other than the fields themselves, it has to be prefixed by **0x**.

offset *OFFSET*

Set an offset which defines where matches of subsequent filters are applied to. Therefore this option is useful only when combined with **link** or a combination of **ht** and **sample**. The offset may be given explicitly by using the **plus** keyword, or extracted from the packet data with **at**. It is possible to mangle the latter using **mask** and/or **shift** keywords. By default, this offset is recorded but not implicitly applied. It is used only to substitute the **nexthdr+** statement. Using the keyword **eat** though inverses this behaviour: the offset is applied always, and **nexthdr+** will fall back to zero.

hashkey *HASHKEY*

Spefify what packet data to use to calculate a hash key for bucket lookup. The kernel adjusts the value according to the hash table's size. For this to work, the option **link** must be given.

classid *CLASSID*

Classify matching packets into the given *CLASSID*, which consists of either 16bit **major** and **minor** numbers or a single 32bit value combining both.

divisor *u32_value*

Specify a modulo value. Used when creating hash tables to define their size or for declaring a **sample** to calculate hash table keys from. Must be a power of two with exponent not exceeding eight.

order *u32_value*

A value to order filters by, ascending. Conflicts with **handle** which serves the same purpose.

sample *SELECTOR*

Used together with **ht** to specify which bucket to add this filter to. This allows one to avoid having to know how exactly the kernel calculates hashes. The additional **divisor** defaults to 256, so must be given for hash tables of different size.

link *HANDLE*

Delegate matching packets to filters in a hash table. *HANDLE* is used to only specify the hash table, so only **htid** may be given, **hash** and **nodeid** have to be omitted. By default, bucket number 0 will be used and can be overridden by the **hashkey** option.

indev *ifname*

Filter on the incoming interface of the packet. Obviously works only for forwarded traffic.

skip_sw

Do not process filter by software. If hardware has no offload support for this filter, or TC offload is not enabled for the interface, operation will fail.

skip_hw

Do not process filter by hardware.

help Print a brief help text about possible options.

SELECTORS

Basically the only real selector is **u32 .** All others merely provide a higher level syntax and are internally translated into **u32 .**

u32 *VAL_MASK_32*
u16 *VAL_MASK_16*
u8 *VAL_MASK_8*

Match packet data to a given value. The selector name defines the sample length to extract (32bits for **u32**, 16bits for **u16** and 8bits for **u8**). Before comparing, the sample is binary AND'ed with the given mask. This way uninteresting bits can be cleared before comparison. The position of the sample is defined by the offset specified in *AT*.

ip *IP*
ip6 *IP6* Assume packet starts with an IPv4 (**ip**) or IPv6 (**ip6**) header. *IP*/*IP6* then allows to match various header fields:

src *ADDR*
dst *ADDR*

Compare Source or Destination Address fields against the value of *ADDR*. The reserved words **default**, **any** and **all** effectively match any address. Otherwise an IP address of the particular protocol is expected, optionally suffixed by a prefix length to match whole subnets. In case of IPv4 a netmask may also be given.

dsfield *VAL_MASK_8*

IPv4 only. Match the packet header's DSCP/ECN field. Synonyms to this are **tos** and **precedence**.

ihl *VAL_MASK_8*

IPv4 only. Match the Internet Header Length field. Note that the value's unit is 32bits, so to match a packet with 24byte header length *u8_value* has to be 6.

protocol *VAL_MASK_8*

Match the Protocol (IPv4) or Next Header (IPv6) field value, e.g. 6 for TCP.

icmp_type *VAL_MASK_8*
icmp_code *VAL_MASK_8*

Assume a next-header protocol of icmp or ipv6-icmp and match Type or Code field values. This is dangerous, as the code assumes minimal header size for IPv4 and lack of extension headers for IPv6.

sport *VAL_MASK_16*
dport *VAL_MASK_16*

Match layer four source or destination ports. This is dangerous as well, as it assumes a suitable layer four protocol is present (which has Source and Destination Port fields right at the start of the header and 16bit in size). Also minimal header size for IPv4 and lack of IPv6 extension headers is assumed.

nofrag
firstfrag
df
mf

IPv4 only, check certain flags and fragment offset values. Match if the packet is not a fragment (**nofrag**), the first fragment (**firstfrag**), if Don't Fragment (**df**) or More Fragments (**mf**) bits are set.

priority *VAL_MASK_8*

IPv6 only. Match the header's Traffic Class field, which has the same purpose and semantics of IPv4's ToS field since RFC 3168: upper six bits are DSCP, the lower two ECN.

flowlabel *VAL_MASK_32*

IPv6 only. Match the Flow Label field's value. Note that Flow Label itself is only 20bytes long, which are the least significant ones here. The remaining upper 12bytes match Version and Traffic Class fields.

tcp *TCPUDP*
udp *TCPUDP*

Match fields of next header of protocol TCP or UDP. The possible values for *TCPDUP* are:

src *VAL_MASK_16*

Match on Source Port field value.

dst *VALMASK_16*

Match on Destination Port field value.

icmp *ICMP*

Match fields of next header of protocol ICMP. The possible values for *ICMP* are:

type *VAL_MASK_8*

Match on ICMP Type field.

code *VAL_MASK_8*

Match on ICMP Code field.

mark *VAL_MASK_32*

Match on netfilter fwmark value.

ether *ETHER*

Match on ethernet header fields. Possible values for *ETHER* are:

src *ether_address AT*
dst *ether_address AT*

Match on source or destination ethernet address. This is dangerous: It assumes an ethernet header is present at the start of the packet. This will probably lead to unexpected things if used with layer three interfaces like e.g. tun or ppp.

EXAMPLES

```
tc filter add dev eth0 parent 999:0 prio 99 protocol ip u32 \
        match ip src 192.168.8.0/24 classid 1:1
```

This attaches a filter to the qdisc identified by **999:0.** It's priority is **99**, which affects in which order multiple filters attached to the same **parent** are consulted (the lower the earlier). The filter handles packets of **protocol** type **ip**, and **match**es if the IP header's source address is within the **192.168.8.0/24** subnet. Matching packets are classified into class **1.1**. The effect of this command might be surprising at first glance:

```
filter parent 1: protocol ip pref 99 u32
filter parent 1: protocol ip pref 99 u32 \
        fh 800: ht divisor 1
filter parent 1: protocol ip pref 99 u32 \
        fh 800::800 order 2048 key ht 800 bkt 0 flowid 1:1 \
        match c0a80800/ffffff00 at 12
```

So parent **1:** is assigned a new **u32** filter, which contains a hash table of size 1 (as the **divisor** indicates). The table ID is **800**. The third line then shows the actual filter which was added above: it sits in table **800** and bucket **0**, classifies packets into class ID **1:1** and matches the upper three bytes of the four byte value at offset **12** to be **0xc0a808**, which is 192, 168 and 8.

Now for something more complicated, namely creating a custom hash table:

```
tc filter add dev eth0 prio 99 handle 1: u32 divisor 256
```

This creates a table of size 256 with handle **1:** in priority **99**. The effect is as follows:

```
filter parent 1: protocol all pref 99 u32
filter parent 1: protocol all pref 99 u32 fh 1: ht divisor 256
filter parent 1: protocol all pref 99 u32 fh 800: ht divisor 1
```

So along with the requested hash table (handle **1:**), the kernel has created his own table of size 1 to hold other filters of the same priority.

The next step is to create a filter which links to the created hash table:

```
tc filter add dev eth0 parent 1: prio 1 u32 \
        link 1: hashkey mask 0x0000ff00 at 12 \
        match ip src 192.168.0.0/16
```

The filter is given a lower priority than the hash table itself so **u32** consults it before manually traversing the hash table. The options **link** and **hashkey** determine which table and bucket to redirect to. In this case the hash key should be constructed out of the second byte at offset 12, which corresponds to an IP packet's third byte of the source address field. Along with the **match** statement, this effectively maps all class C networks below 192.168.0.0/16 to different buckets of the hash table.

Filters for certain subnets can be created like so:

```
tc filter add dev eth0 parent 1: prio 99 u32 \
                ht 1: sample u32 0x00000800 0x0000ff00 at 12 \
                match ip src 192.168.8.0/24 classid 1:1
```

The bucket is defined using the **sample** option: In this case, the second byte at offset 12 must be 0x08, exactly. In this case, the resulting bucket ID is obviously 8, but as soon as **sample** selects an amount of data which could exceed the **divisor**, one would have to know the kernel-internal algorithm to deduce the destination bucket. This filter's **match** statement is redundant in this case, as the entropy for the hash key does not exceed the table size and therefore no collisions can occur. Otherwise it's necessary to prevent matching unwanted packets.

Matching upper layer fields is problematic since IPv4 header length is variable and IPv6 supports extension headers which affect upper layer header offset. To overcome this, there is the possibility to specify **nexthdr+** when giving an offset, and to make things easier there are the **tcp** and **udp** matches which use **nexthdr+** implicitly. This offset has to be calculated in beforehand though, and the only way to achieve that is by doing it in a separate filter which then links to the filter which wants to use it. Here is an example of doing so:

```
tc filter add dev eth0 parent 1:0 protocol ip handle 1: \
        u32 divisor 1
tc filter add dev eth0 parent 1:0 protocol ip \
        u32 ht 1: \
        match tcp src 22 FFFF \
        classid 1:2
tc filter add dev eth0 parent 1:0 protocol ip \
        u32 ht 800: \
        match ip protocol 6 FF \
        match ip firstfrag \
        offset at 0 mask 0f00 shift 6 \
        link 1:
```

This is what is being done: In the first call, a single element sized hash table is created so there is a place to hold the linked to filter and a known handle (**1:**) to reference to it. The second call then adds the actual filter, which pushes packets with TCP source port 22 into class **1:2**. Using **ht**, it is moved into the hash table created by the first call. The third call then does the actual magic: It matches IPv4 packets with next layer protocol 6 (TCP), only if it's the first fragment (usually TCP sets DF bit, but if it doesn't and the packet is fragmented, only the first one contains the TCP header), and then sets the offset based on the IP header's IHL field (right-shifting by 6 eliminates the offset of the field and at the same time converts the value into byte unit). Finally, using **link**, the hash table from first call is referenced which holds the filter from second call.

SEE ALSO

tc(8),
cls_u32.txt at **http://linux-tc-notes.sourceforge.net/**