NAME

ng_tag - mbuf tags manipulating netgraph node type

SYNOPSIS

#include <netgraph/ng_tag.h>

DESCRIPTION

The **tag** node type allows mbuf packet tags (see mbuf_tags(9)) to be examined, stripped or applied to data travelling through a Netgraph network. Mbuf tags are used in many parts of the FreeBSD kernel network subsystem, including the storage of VLAN tags as described in vlan(4), Mandatory Access Control (MAC) labels as described in mac(9), IPsec policy information as described in ipsec(4), and packet filter tags used by pf(4). One should also consider useful setting or checking ipfw(8) tags, which are implemented as mbuf tags, too.

Each node allows an arbitrary number of connections to arbitrarily named hooks. With each hook is associated a tag which will be searched in the list of all tags attached to a packet incoming to this hook, a destination hook for matching packets, a destination hook for non-matching packets, a tag which will be appended to data leaving node through this hook, and various statistics counters.

The list of incoming packet's tags is traversed to find a tag with specified *type* and *cookie* values. Upon match, if specified *tag_len* is non-zero, *tag_data* of tag is checked to be identical to that specified in the hook structure. Packets with matched tags are forwarded to "match" destination hook, or forwarded to "non-match" hook otherwise. Either or both destination hooks can be an empty string, or may not exist, in which case the packet is dropped.

Tag list of packets leaving the node is extended with a new tag specified in outgoing hook structure (it is possible to avoid appending a new tag to pass packet completely unchanged by specifying zero *type* and *cookie* values in the structure of the corresponding outgoing hook). Additionally, a tag can be stripped from incoming packet after match if *strip* flag is set. This can be used for simple tag removal or tag replacement, if combined with tag addition on outgoing matching hook. Note that new tag is appended unconditionally, without checking if such a tag is already present in the list (it is up to user to check if this is a concern).

New hooks are initially configured to drop all incoming packets (as all hook names are empty strings; zero values can be specified to forward all packets to non-matching hook), and to forward all outgoing packets without any tag appending.

Data payload of packets passing through the node is completely unchanged, all operations can affect tag list only.

HOOKS

This node type supports any number of hooks having arbitrary names. In order to allow internal optimizations, user should never try to configure a hook with a structure pointing to hooks which do not exist yet. The safe way is to create all hooks first, then begin to configure them.

CONTROL MESSAGES

This node type supports the generic control messages, plus the following:

NGM_TAG_SET_HOOKIN (**sethookin**)

This command sets tag values which will be searched in the tag list of incoming packets on a hook. The following structure must be supplied as an argument:

```
struct ng_tag_hookin {
  char          thisHook[NG_HOOKSIZ];    /* name of hook */
  char          ifMatch[NG_HOOKSIZ];      /* match dest hook */
  char          ifNotMatch[NG_HOOKSIZ];  /* !match dest hook */
  uint8_t   strip;          /* strip tag if found */
  uint32_t  tag_cookie;             /* ABI/Module ID */
  uint16_t  tag_id;         /* tag ID */
  uint16_t  tag_len;        /* length of data */
  uint8_t   tag_data[0];            /* tag data */
};
```

The hook to be updated is specified in *thisHook*. Data bytes of tag corresponding to specified *tag_id* (type) and *tag_cookie* are placed in the *tag_data* array; there must be *tag_len* of them. Matching and non-matching incoming packets are delivered out the hooks named *ifMatch* and *ifNotMatch*, respectively. If *strip* flag is non-zero, then found tag is deleted from list of packet tags.

NGM_TAG_GET_HOOKIN (**gethookin**)

This command takes an ASCII string argument, the hook name, and returns the corresponding *struct ng_tag_hookin* as shown above.

NGM_TAG_SET_HOOKOUT (**sethookout**)

This command sets tags values which will be applied to outgoing packets. The following structure must be supplied as an argument:

```
struct ng_tag_hookout {
  char          thisHook[NG_HOOKSIZ];    /* name of hook */
  uint32_t  tag_cookie;             /* ABI/Module ID */
  uint16_t  tag_id;         /* tag ID */
```

```
    uint16_t  tag_len;          /* length of data */
    uint8_t   tag_data[0];              /* tag data */
};
```

The hook to be updated is specified in *thisHook*. Other variables mean basically the same as in *struct ng_tag_hookin* shown above, except used for setting values in a new tag.

NGM_TAG_GET_HOOKOUT (**gethookout**)

This command takes an ASCII string argument, the hook name, and returns the corresponding *struct ng_tag_hookout* as shown above.

NGM_TAG_GET_STATS (**getstats**)

This command takes an ASCII string argument, the hook name, and returns the statistics associated with the hook as a *struct ng_tag_hookstat*.

NGM_TAG_CLR_STATS (**clrstats**)

This command takes an ASCII string argument, the hook name, and clears the statistics associated with the hook.

NGM_TAG_GETCLR_STATS (**getclrstats**)

This command is identical to NGM_TAG_GET_STATS, except that the statistics are also atomically cleared.

Note: statistics counters as well as three statistics messages above work only if code was compiled with the NG_TAG_DEBUG option. The reason for this is that statistics is rarely used in practice, but still consumes CPU cycles for every packet. Moreover, it is even not accurate on SMP systems due to lack of synchronization between threads, as this is very expensive.

SHUTDOWN

This node shuts down upon receipt of a NGM_SHUTDOWN control message, or when all hooks have been disconnected.

EXAMPLES

It is possible to do a simple L7 filtering by using ipfw(8) tags in conjunction with ng_bpf(4) traffic analyzer. Example below explains how to filter DirectConnect P2P network data traffic, which cannot be done by usual means as it uses random ports. It is known that such data connection always contains a TCP packet with 6-byte payload string "$Send|". So ipfw's **netgraph** action will be used to divert all TCP packets to an ng_bpf(4) node which will check for the specified string and return non-matching packets to ipfw(8). Matching packets are passed to **ng_tag** node, which will set a tag and pass them back to ng_bpf(4) node on a hook programmed to accept all packets and pass them back to ipfw(8). A script

provided in ng_bpf(4) manual page will be used for programming node. Note that packets diverted from ipfw(8) to Netgraph have no link-level header, so offsets in tcpdump(1) expressions must be altered accordingly. Thus, there will be expression "ether[40:2]=0x244c && ether[42:4]=0x6f636b20" on incoming hook and empty expression to match all packets from **ng_tag**.

So, this is ngctl(8) script for nodes creating and naming for easier access:

```
/usr/sbin/ngctl -f- <<-SEQ
        mkpeer ipfw: bpf 41 ipfw
        name ipfw:41 dcbpf
        mkpeer dcbpf: tag matched th1
        name dcbpf:matched ngdc
SEQ
```

Now "ngdc" node (which is of type **ng_tag**) must be programmed to echo all packets received on the "th1" hook back, with the ipfw(8) tag 412 attached. MTAG_IPFW value for *tag_cookie* was taken from file <*netinet/ip_fw.h*> and value for *tag_id* is tag number (412), with zero tag length:

```
ngctl msg ngdc: sethookin { thisHook=\"th1\" ifNotMatch=\"th1\" }
ngctl msg ngdc: sethookout { thisHook=\"th1\" \
  tag_cookie=1148380143 \
  tag_id=412 }
```

Do not forget to program ng_bpf(4) "ipfw" hook with the above expression (see ng_bpf(4) for script doing this) and "matched" hook with an empty expression:

```
ngctl msg dcbpf: setprogram { thisHook=\"matched\" ifMatch=\"ipfw\" \
  bpf_prog_len=1 bpf_prog=[ { code=6 k=8192 } ] }
```

After finishing with netgraph(4) nodes, ipfw(8) rules must be added to enable packet flow:

```
ipfw add 100 netgraph 41 tcp from any to any iplen 46
ipfw add 110 reset tcp from any to any tagged 412
```

Note: one should ensure that packets are returned to ipfw after processing inside netgraph(4), by setting appropriate sysctl(8) variable:

```
sysctl net.inet.ip.fw.one_pass=0
```

SEE ALSO

netgraph(4), ng_bpf(4), ng_ipfw(4), ipfw(8), ngctl(8), mbuf_tags(9)

HISTORY

The **ng_tag** node type was implemented in FreeBSD 6.2.

AUTHORS

Vadim Goncharov <*vadimnuclight@tpu.ru*>

BUGS

For manipulating any tags with data payload (that is, all tags with non-zero *tag_len*) one should care about non-portable machine-dependent representation of tags on the low level as byte stream. Perhaps this should be done by another program rather than manually.