Netfilter connection tracking and nat helper modules

Harald Welte, laforge@gnumonks.org
Version 1.2, 2000/10/14 20:37:53

Well... This initially wasn't intended to be a publicly available document. It just contains some of my thoughts during a summer holiday in italy, where I was trying to port the ip_masq_irc module to the new netfilter conntrack/nat framework. There was no documentation about this topic available on the net, so I tried to understand the netfilter code.

Contents

1 Introduction

1.1 What the hell is this all about?

This document gives a brief description how to write netfilter connection tracking helper and nat helper modules. Netfilter is the packet filtering / NAT infrastructure provided by the Linux 2.4.x kernel.

1.2 What to read first

I strongly recommend You reading Rusty Russel's 'netfilter hacking howto' which is available from the netfilter project homepage at http://netfilter.kernelnotes.org

2 Connection tracking helper modules

2.1 Description

The duty of a connection tracking module is to specify which packtets belong to an already established connection. The module has the following means to do that:

- Tell netfilter which packets our module is interested in
- Register a conntrack function with netfilter. This function is called for every "interesting" packet (as decided by the callback function above)
- Call ip_conntrack_expect_related to tell netfilter which packets are related to the connection.

2.2 Structures and Functions available

At first some basic structures

'struct ip_conntrack_tuple' (just printed the fields valid for TCP)

src.ip
: the source IP address

src.u.tcp.port
: the TCP source port

dst.ip
: the destination IP address

dst.protonum
: the protocol (IPPROTO_TCP, ...)

dst.u.tcp.port
: the TCP destination port

Your kernel module's init function has to call 'ip_conntrack_helper_register()' with a pointer to a 'struct ip_conntrack_helper'. This struct has the following fields:

list
: This is the header for the linked list. Netfilter handles this list internally. Just initialize it with { NULL, NULL }

tuple
: This is a 'struct ip_conntrack_tuple' which specifies the packets our conntrack helper module is interested in.

mask
: Again a 'struct ip_conntrack_tuple'. This mask specifies which bits of `tuple` are valid.

help
: The function which netfilter should call for each packet matching tuple+mask

2.3 Example skeleton of a conntrack helper module

```
#define FOO_PORT        111

static int foo_help(const struct iphdr *iph, size_t len,
                struct ip_conntrack *ct,
                enum ip_conntrack_info ctinfo)
{
        /* analyze the data passed on this connection and
           decide how related packets will look like */

        if (there_will_be_new_packets_related_to_this_connection)
        {
                t = new_tuple_specifying_related_packets;
                ip_conntrack_expect_related(ct, &t);

                /* save information important for NAT in
                        ct->help.ct_foo_info;   */

        }
        return NF_ACCEPT;
}

static struct ip_conntrack_helper foo;

static int __init init(void)
{
        memset(&foo, 0, sizeof(struct ip_conntrack_helper);

        /* we are interested in all TCP packets with destport 111 */
        foo.tuple.dst.protonum = IPPROTO_TCP;
        foo.tuple.dst.u.tcp.port = htons(FOO_PORT);
        foo.mask.dst.protonum = 0xFFFF;
        foo.mask.dst.u.tcp.port = 0xFFFF;
        foo.help = foo_help;

        return ip_conntrack_helper_register(&foo);
}

static void __exit fini(void)
{
        ip_conntrack_helper_unregister(&foo);
}
```

3 NAT helper modules

3.1 Description

NAT helper modules do some application specific NAT handling. Usually this includes on-the-fly manipulation of data. Think about the PORT command in FTP, where the client tells the server which ip/port to connect to. Thererfore a FTP helper module has to replace the ip/port after the PORT command in the FTP control connection.

If we are dealing with TCP, things get slightly more complicated. The reason is a possible change of the packet size (FTP example: The length of the string representing an IP/port tuple after the PORT command has changed). If we had to change the packet size, we have a syn/ack difference between left and right side of the NAT box. (i.e. if we had extended one packet by 4 octets, we have to add this offset to the TCP sequence number of each following packet)

Special NAT handling of all related packets is required, too. Take as example again FTP, where all incoming packets of the DATA connection have to be NATed to the ip/port given by the client with the PORT command on the control connection.

- callback for the packet causing the related connection (foo_help)
- callback for all related packets (foo_nat_expected)

3.2 Structures and Functions available

Your nat helper module's 'init()' function has to call 'ip_nat_helper_register()' with a pointer to a 'struct ip_nat_helper'. This struct hast the following members:

list

Just again the list header for netfilters internal use. Initialize this with { NULL, NULL }.

tuple

a 'struct ip_conntrack_tuple' describing which packets our nat helper is interested in.

mask

a 'struct ip_conntrack_tuple', telling netfilter which bits of `tuple` are valid.

help

The help function which is called for each packet matching tuple+mask.

name

The uniqe name this nat helper is identified by.

3.3 Example NAT helper module

```
#define FOO_PORT        111

static int foo_nat_expected(struct sk_buff **pksb,
                        unsigned int hooknum,
                        struct ip_conntrack *ct,
                        struct ip_nat_info *info,
                        struct ip_conntrack *master,
                        struct ip_nat_info *masterinfo,
                        unsigned int *verdict)
```

```
/* called whenever a related packet (as specified in the connectiontracking
   module) arrives
   params:      pksb    packet buffer
                hooknum HOOK the call comes from (POST_ROUTING, PRE_ROUTING)
                ct      information about this (the related) connection
                info    (STATE: established, related, new)
                master  information about the master connection
                masterinfo      (STATE: established, related, new) of master */
{
        /* basically just change ip/port of the packet to the masqueraded
           values (read from master->tuplehash) */

}

static int foo_help(struct ip_conntrack *ct,
                struct ip_nat_info *info,
                enum ip_conntrack_info ctinfo,
                unsigned int hooknum,
                struct sk_buff  **pksb)
/* called for the packet causing related packets
   params:      ct      information about tracked connection
                info    (STATE: related, new, established, ... )
                hooknum HOOK the call comes from (POST_ROUTING, PRE_ROUTING)
                pksb    packet buffer
*/
{
        /* extract information about future related packets,
           exchange address/port with masqueraded values,
           insert tuple about related packets */
}

static struct ip_nat_expect foo_expect = {
        { NULL, NULL },
        foo_nat_expected };

static struct ip_nat_helper hlpr;

static int __init(void)
{
        if (ip_nat_expect_register(&foo_expect))
        {
                memset(&hlpr, 0, sizeof(struct ip_nat_helper));
                hlpr.list = { NULL, NULL };
                hlpr.tuple.dst.protonum = IPPROTO_TCP;
                hlpr.tuple.dst.u.tcp.port = htons(FOO_PORT);
                hlpr.mask.dst.protonum = 0xFFFF;
                hlpr.mask.dst.u.tcp.port = 0xFFFF;
                hlpr.help = foo_help;

                ip_nat_helper_register(hlpr);
        }
```

```
}

static void __exit(void)
{
        ip_nat_expect_unregister(&foo_expect);
        ip_nat_helper_unregister(&hlpr);
}
```

4 Credits

I want to thank all the great netfilter folks, especially Rusty Russel, for providing us (the Linux community) with this neat infrastructure.