The netfilter framework in Linux 2.4

Harald Welte laforge@gnumonks.org $Date: 2000/09/24 13:20:24 $

This is the paper on which my talk about netfilter at Linux-Kongress 2000 is based. It describes the netfilter infrastructure, as well as the systems for packet filtering, NAT and packet mangling on top of it

Contents

1 PART I - Netfilter basics / concepts

1.1 What is netfilter?

Netfilter is definitely more than any of the firewall subsystems in the past linux kernels. Netfilter provides a abstract, generalized framework of which one particular incarnation is the packet filtering subsystem. So don't expect a talk about "how to set up a firewall or a masquerading gateway in 2.4". This would only cover a part of netfilter.

The netfilter framework consists out of three parts:

1. Each protocol defines a set of 'hooks' (IPv4 defines 5), which are well-defined points in a packet's traversal of that protocol stack. At each of these points, the protocol stack will call the netfilter framework with the packet and the hook number.
2. Parts of the kernel can register to listen to the different hooks for each protocol. So when a packet is passed to the netfilter framework, it checks to see if anyone has registered for that protocol and hook; if so, they get a chance to examine (and possibly alter) the packet, discard it, allow it to pass or ask netfilter to queue the packet for userspace.
3. Packets that have been queued are collected for sending to userspace; these packets are handled asynchronously. A userspace process can examine the packet, can alter it, and reinject it at the same hook it left the kernel.

All the packet filtering / NAT / ... stuff is based on this framework. There is no more dirty packet altering code spread all over the network stack.

The netfilter framework currently has been implemented for IPv4, IPv6 and DECnet.

1.2 Why did we need netfilter?

This chapter could be called 'What is wrong with ipchains?', too. So why did we need this change? (I only give a few examples here)

- No infrastructure for passing packets to userspace, so all code which does some packet fiddling must be done as kernel code. Kernel programming is hard, must be done in C, and is dangerous.
- Transparent proxying is extremely difficult We have to look up _every_ packet to see if there's a socket bound to that adderess. No clean interface, 34 #ifdef' in 11 different files of the network stack
- Creating of packet filter rules independent of interface address is impossible. We must know local interface address to distinguish locally-generated or locally-terminated packets from through packets. The forward chain has only information on outgoing interface. So we must try to figure out where the packet came from.
- Masquerading and packet filtering are implemented as one part This makes the firewalling code way too complex.
- Ipchains code is neither modular nor extensible (eg. for MAC adress filtering)

1.3 The authors of netfilter

The concept of the netfilter framework and most of its implementation were done by Rusty Russell. He is co-author if ipchains and is the current Linux Kernel IP firewall maintainer. Rusty got paid one Year by Watchguard (a firewall company) to do nothing, so he had enough time to do it :)

The official netfilter core team consists out of Rusty Russell, Marc Boucher and James Morris. Of course there are various other hackers who have contributed some stuff (like me *g*)

1.4 Netfilter architecture in IPv4

A Packet Traversing the Netfilter System:

```
--->[1]--->[ROUTE]--->[3]--->[4]--->
              |            ^
              |            |
              |         [ROUTE]
              v            |
             [2]          [5]
              |            ^
              |            |
              v            |
```

Packets come in from the left. After verification of the IP checksum, the packets hit the NF_IP_PRE_ROUTING [1] hook.

Next they enter the routing code, which decides if the packets are local or have to be passed to another interface.

If the packets are considered to be local, they traverse th NF_IP_LOCAL_IN [2] hook and get passed to the process (if any) afterwards.

If the packets are routed to another interface, they pass the NF_IP_FORWARD [3] hook.

The packet passes a final netfilter hook, NF_IP_POST_ROUTING [4], before they get transmitted on the target interface.

The NF_IP_LOCAL_OUT [5] hook is called for locally generated packets. Here You can see that routing occurs after this hook is called: in fact, the routing code is called first (to figure out the source IP address and some IP options), and called again if the packet is altered.

Locally generated packets hit NF_IP_POST_ROUTING [4], too.

1.5 Netfilter base

Now we have an example of netfilter IPv4, you can see how each hook is activated.

Kernel modules can register for one or more of this hook and get called for each packet traversing the hook. The module is free to alter the packet and returns netfilter one of these values:

- NF_ACCEPT continue traversal as normal

- NF_DROP drop the packet; do not continue traversal
- NF_STOLEN I've taken over the packet; do not continue traversal
- NF_QUEUE queue the packet (usually for userspace handling)
- NF_REPEAT call this hook again

1.6 Packet selection: IP tables

A packet selection system called IP tables has been built based on the netfilter framework. It is a direct descendant of ipchains, with extensibility.

Kernel modules can register a new table, and ask for a packet to traverse a given table. This packet selection is used for packet filtering (the 'filter' table), Network Address Translation (the 'nat' table) and general packet mangling (the 'mangle' table).

The three big parts of Linux 2.4 packet handling are built using netfilter hooks and IP tables. They are seperate modules and are independent from each other. They all plug in nicely in the infrastructure provided by netfilter.

1. Packet filtering This table 'filter' should never alter packets, only filter them. One of the advantages of iptables over ipchains is that it is small and fast, and it hooks into netfilter at the NF_IP_LOCAL_IN, NF_IP_FORWARD and NF_IP_LOCAL_OUT hooks.

   Therefore, for each packet there is one, and only one, place to filter it. This is one big change compared to ipchains, where a forwarded packet used to traverse three chains.

2. NAT The nat table listens at three netfilter hooks: NF_IP_PRE_ROUTING and NF_IP_POST_ROUTING to do source and destination NAT for routed packets. For destination altering of local packets, the NF_IP_LOCAL_OUT hook is used.

   This table is different from the 'filter' table, in that only the first packet of a new connection will traverse the table. The result of this traversal is then applied to all future packets of the same connection.

   The NAT table is used for source NAT, destination NAT, masquerading (which is a special case of source nat) and transparent proxying (which is a special case of destination nat).

3. Packet mangling The 'mangle' table registers at the NF_IP_PRE_ROUTING and NF_IP_LOCAL_OUT hooks.

   Using the mangle table You can modify the packet itself or some of the out-of-band data attached to the packet. Currently the alteration of the TOS bits as well as setting the nfmark field inside the skb is implemented on top of the mangle table.

1.7 Connection tracking

Connection tracking is fundamental to NAT, but has been implemented as a seperate module. This allows an extension to the packet filtering code to simply use connection tracking for "stateful firewalling". (the 'state' match)

2 PART II - packet filtering using iptables and netfilter

2.1 Overview

I expect You are familiar with TCP/IP, routing, firewall concepts and packet filtering in general.

As already explained in Part I, the filter table listens on three hooks, thus providing us three chains for packet filtering.

All packets coming from the network and destined for the local box traverse the INPUT chain.

All packets which are forwarded (routed) by us traverse the FORWARD chain (and only the FORWARD chain). Please again note this difference to the previous linux firewall implementations!

Finally, the packets originating from the local box traverse the OUTPUT chain.

2.2 Inserting rules into chains

To insert/delete/modify any rules in linux 2.4 IP tables we have a neat and powerful commandline tool, called 'iptables'. I don't want to get too deep into all its features and extensibility. Here are some of its major features:

- It handles all different kinds of IP tables. Currently the filter, nat and mangle tables, but also all future table modules
- It supports plugins for new matches and new targets. Thus, nobody ever needs to patch anything to provide a netfilter extension. You have a kernel module doing the real work and a iptables plugin (dynamic library) to add the neccessary configuration options.
- It comes in two incarnations: iptalbes (IPv4) and ip6tables (IPv6). Both of them are based on the same library and mostly the same code.

2.2.1 Basic iptables commands

An iptables command usually consists out of 5 parts:

1. which table we want to work with
2. which chain in this table we want it to use
3. an operation (insert, add, delete, modify)
4. a target for this particular rule
5. a description of which packets we want to match this rule

The basic syntax is

```
iptables -t table -Operation chain -j target match(es)
```

To add a rule allowing all traffic from anywhere to our local smtp port:

```
iptables -t filter -A INPUT -j ACCEPT -p tcp --dport smtp
```

Of course there are various other commands like flush chain, set the default policy of a chain, add a user-defined chain, ...

Basic Operations:

```
-A      append rule
-I      insert rule
-D      delete rule
-R      replace rule
-L      list rules
```

Basic Targets, common to all chains:

```
ACCEPT  accept the packet
DROP    drop the packet
QUEUE   queue packet to userspace
RETURN  return to the previous (calling) chain
foobar  user defined chain
```

Basic matches, common to all chains:

```
-p      protocol (tcp/icmp/udp/...)
-s      source address (ip address/masklen)
-d      destination address (ip address/masklen)
-i      incoming interface
-o      outgoing interface
```

Apart from these basic operations, matches and targets there are various extensions, which I'll describe in the apropriate chapters.

2.3 iptables match extensions for filtering

There are various extensions which are useful for packet filtering. Describing them all in detail would take way too much time. Just to give You an impression about the power :)

At first there are some match extensions, which give us more power to describe which packets to match:

- TCP match extensions to match source port, destination port, arbitrary combinations of TCP flags, tcp options.
- UPD match extensions to match source port and destination port
- ICMP match extension to match icmp type
- MAC match extension to match incoming mac (ethernet) address
- MARK match extension to match the nfmark

- OWNER match extension (for locally generated packets only) to match user id, group id, process id, session id
- LIMIT match extension to match only a certain limit of packets per time frame. Very useful to prevent forwarding of any kind of flooding.
- STATE match extension to match packets of a certain state (decided by the connection tracking subsystem). Possible states are
  - INVALID (not matched against a connection),
  - ESTABLISHED (packet belongs to an already established connection),
  - NEW (packet would establish a new connection) and
  - RELATED (packet is in some way related to an already established connection. For example an ICMP error message or a ftp data connection)

TOS match extension to match the value of the TOS IP header field.

2.4 iptables target extensions for filtering

- LOG log matched packets via syslog()
- ULOG log matched packets via userspace logging daemon (supports interpreter and output plugins for flexible logging)
- REJECT not only drop the packet, but also send some kind of error message to the sender (which message is configurable)
- MIRROR retransmit the packet after exchanging source and destination IP address

3 PART III - NAT using iptables and netfilter

Regarding to NAT (Network Address Translation) the previous Linux Kernels only supported one spacial case called "Masquerading"

Netfilter now enables Linux to do any kind of NAT.

Nat is divided into 'source NAT' and 'destination NAT'.

Source NAT alters the source address of a packet while passing the NF_IP_POST_ROUTING hook. Masquerading is a special application of SNAT

Destination NAT alters the destination address of a packet while passing the NF_IP_LOCAL_OUT respectively NF_IP_PRE_ROUTING hook. Port forwarding and transparent proxying are forms of DNAT.

3.1 iptables target extensions for NAT

SNAT

Change the source address to something different

Example:

```
iptables -t nat -A POSTROUTING -j SNAT --to-source 1.2.3.4
```

MASQUERADE

SNAT for dialup connections with dynamic ip address

Does almost the same as SNAT, but if the link goes down, all connection tracking information is dropped. The connections are lost anyway, because we get a different IP address at reconnect.

Example:

```
iptables -t nat -A POSTROUTING -j MASQUERADE -o ppp0
```

DNAT

Change the destination address to something different

This is done at the PREROUTING chain, just as the packet comes in. Therefore, anything else on the Linux box itself (routing, packet filtering) will se the packet to its real (new) destination.

Example:

```
iptables -t nat -A PREROUTING -j DNAT --to-destination 1.2.3.4:8080 -p tcp --dport 80 -i eth1
```

REDIRECT

Redirect packets to local destination

Exactly the same as doing DNAT to the address of the incoming interface

Example:

```
iptables -t nat -A PREROUTING -j REDIRECT --to-port 3128 -i eth1 -p tcp --dport 80
```

4 PART IV - Packet mangling using iptables and netfilter

The 'mangle' table enables us to alter the packet itself or some data accompaning the packet.

4.1 iptables target extensions for packet mangling

MARK

set the value of the nfmark field

We can change the value of the nfmark field. The nfmark is just a user defined mark (anything within the range of an unsigned long) of the packet. The mark value is used to do policy routing, tell ipqmpd (the userspace queue multiplex daemon) which process to queue the packet to, etc.

Example: iptables -t mangle -A PREROUTING -j MARK –set-mark 0x0a -p tcp

TOS

set the value of the TOS bits inside the IP header.

We can change the value of the type of service bits inside the IP haeder. This is useful if You are using TOS based packet scheduling / routing.

Example: iptables -t mangle -A PREROUTING -j TOS –set-tos 0x10 -p tcp –dport ssh

5 Queueing packets to userspace

As I already mentioned, at any time in any netfilter chain, the packet can be queued to userspace. The actual queuing is done by a kernel module (ip_queue.o).

The packets (including metadata like nfmark and mac address) are sent to an userspace process using netlink sockets. This process can do whatever it wants to do with the packet.

After the userspace process is done with its work on the packet, it can either reinject the packet into the kernel, or set a verdict (DROP, ...) what to do with the packet.

This is one key technology of netfilter, enabling to do complicated packet handling by userspace processes. Thus, preventing more complexity in the kernel space.

Userspace packet handling processes can be easily developed using a netfilter-provided library called 'libipq'.

Currently only one userspace process is supported, but the first beta release of an userspace ip queueing multiplex daemon (ipqmpd) is available. ipqmpd provides a compatibility library (libipqmpd) which makes upgrading from raw ipqueue interface to the new ipqpmd as easy as relinking to another library.

6 PART V Credits

Credits to all the netfilter hackers, especially the core team.

Namely: **Paul 'Rusty' Russel**, **Marc Boucher** and **James Morris**.

Additional special thanks to Rusty for his 'netfilter-hacking-HOWTO', 'packet-filtering-HOWTO' and 'NAT-HOWTO' which I heavily used as a basis for this presentation.