Linux 2.4.x netfilter/iptables firewalling internals

Harald Welte
laforge@gnumonks.org

25. April 2002

1 Introduction

The Linux 2.4.x kernel series has introduced a totally new kernel firewalling subsystem. It is much more than a plain successor of ipfwadm or ipchains.

The netfilter/iptables project has a very modular design and it's sub-projects can be split in several parts: netfilter, iptables, connection tracking, NAT and packet mangling.

While most users will already have learned how to use the basic functions of netfilter/iptables in order to convert their old ipchains firewalls to iptables, there's more advanced but less used functionality in netfilter/iptables.

The presentation covers the design principles behind the netfilter/iptables implementation. This knowledge enables us to understand how the individual parts of netfilter/iptables fit together, and for which potential applications this is useful.

2 Internal netfilter/iptables architecture

2.1 Netfilter hooks in protocol stacks

One of the major motivations behind the redesign of the linux packet filtering and NAT system during the 2.3.x kernel series was the widespread firewall specific code parts within the core IPv4 stack. Ideally the core IPv4 stack (as used by regular hosts and routers) shouldn't contain any firewalling specific code, resulting in no unwanted interaction and less code complexity. This desire lead to the invention of *netfilter*.

2.1.1 Architecture of netfilter

Netfilter is basically a system of callback functions within the network stack. It provides a non-portable API towards in-kernel networking extensions.

What we call *netfilter hook* is a well-defined call-out point within a layer three protocol stack, such as IPv4, IPv6 or DECnet. Any layer three network stack can define an arbitrary number of hooks, usually placed at strategic points within the packet flow.

Any other kernel code can now subsequently register callback functions for any of these hooks. As in most sytems will be more than one callback function registered for a particular hook, a *priority* is specified upon registration of the callback function. This priority defines the order in which the individual callback functions at a particular hook are called.

The return value of any registered callback functions can be:

- **NF_ACCEPT**: continue traversal as usual
- **NF_DROP**: drop the packet; do not continue traversal
- **NF_STOLEN**: callback function has taken over the packet; do not continue

- **NF_QUEUE**: enqueue the packet to userspace
- **NF_REPEAT**: call this hook again

2.1.2 Netfilter hooks within IPv4

The IPv4 stack provides five netfilter hooks, which are placed at the following peculiar places within the code:

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

             local processes
```

Packets received on any network interface arrive at the left side of the diagram. After the verification of the IP header checksum, the NF_IP_PRE_ROUTING [1] hook is traversed.

If they "survive" (i.e. NF_ACCEPT is returned), the packet enters the routing code. Where we continue from here depends on the destintion of the packet.

Packets with a local destination (i.e. packets where the destination address is one of the own IP addresses of the host) traverse the NF_IP_LOCAL_IN [2] hook. If all callback function return NF_ACCEPT, the packet is finally passed to the socket code, which eventually passes the packet to a local process.

Packets with a remote destination (i.e. packets which are forwarded by the local machine) traverse the NF_IP_FORWARD [3] hook. If they "survive", they finally pass the NF_IP_POST_ROUTING [4] hook and are sent off the outgoing network interface.

Locally generated packets first traverse the NF_IP_LOCAL_OUT [5] hook, then enter the routing code, and finally go through the NF_IP_POST_ROUTING [4] hook before being sent off the outgoing network interface.

2.1.3 Netfilter hooks within IPv6

As the IPv4 and IPv6 protocols are very similar, the netfilter hooks within the IPv6 stack are placed at exactly the same locations as in the IPv4 stack. The only change are the hook names: NF_IP6_PRE_ROUTING, NF_IP6_LOCAL_IN, NF_IP6_FORWARD, NF_IP6_POST_ROUTING, NF_IP6_LOCAL_OUT.

2.1.4 Netfilter hooks within DECnet

There are seven decnet hooks. The first five hooks (NF_DN_PRE_ROUTING, NF_DN_LOCAL_IN, NF_DN_FORWARD, NF_DN_LOCAL_OUT, NF_DN_POST_ROUTING) are prrетty much the same as in IPv4. The last two hooks (NF_DN_HELLO, NF_DN_ROUTE) are used in conjunction with DECnet Hello and Routing packets.

2.1.5 Netfilter hooks within ARP

Recent kernels[1] have added support for netfilter hooks within the ARP code. There are two hooks: NF_ARP_IN and NF_ARP_OUT, for incoming and outgoing ARP packets respectively.

[1] IIRC, starting with 2.4.19-pre3

2.1.6 Netfilter hooks within IPX

There have been experimental patches to add netfilter hooks to the IPX code, but they never got integrated into the kernel source.

2.2 Packet selection using IP Tables

The IP tables core (ip_tables.o) provides a generic layer for evaluation of rulesets.

An IP table consists out of an arbitrary number of *chains*, which in turn consist out of a linear list of *rules*, which again consist out of any number of *matches* and one *target*.

Chains can further be devided into two classes: Either *builtin chains* or *user-defined chains*. Builtin chains are always present, they are created upon table registration. They are also the entry points for table iteration. User defined chains are created at runtime upon user interaction.

Matches specify the matching criteria, there can be zero or more matches

Targets specify the action which is to be executed in case **all** matches match. There can only be a single target per rule.

Matches and targets can either be *builtin* or *linux kernel modules*.

There are two special targets:

- By using a chain name as target, it is possible to jump to the respective chain in case the matches match.
- By using the RETURN target, it is possible to return to the previous (calling) chain

The IP tables core handles the following functions

- Registering and unregistering tables
- Registering and unregistering matches and targets (can be implemented as linux kernel modules)
- Kernel / userspace interface for manipulation of IP tables
- Traversal of IP tables

2.2.1 Packet filtering unsing the “filter” table

Traditional packet filtering (i.e. the successor to ipfwadm/ipchains) takes place in the “filter” table. Packet filtering works like a sieve: A packet is (in the end) either dropped or accepted - but never modified.

The “filter” table is implemented in the *iptable_filter.o* module and contains three builtin chains:

- **INPUT** attaches to NF_IP_LOCAL_IN
- **FORWARD** attaches to NF_IP_FORWARD
- **OUTPUT** attaches to NF_IP_LOCAL_OUT

The placement of the chains / hooks is done in such way, that evey concievable packet always traverses only one of the built-in chains. Packets destined for the local host traverse only INPUT, packets forwarded only FORWARD and locally-originated packets only OUTPUT.

2.2.2 Packet mangling using the “mangle” table

As stated above, operations which would modify a packet do not belong in the “filter” table. The “mangle” table is available for all kinds of packet manipulation - but not manipulation of addresses (which is NAT).

The mangle table attaches to all five netfilter hooks and provides the respectiva builtin chains (PREROUTING, INPUT, FORWARD, OUTPUT, POSTROUTING) [2].

[2] This has changed through recent 2.4.x kernel series, old kernels may only support three (PREROUTING, POSTROUTING, OUTPUT) chains.

2.3 Connection Tracking Subsystem

Traditional packet filters can only match on matching criteria within the currently processed packet, like source/destination IP address, port numbers, TCP flags, etc. As most applications have a notion of connections or at least a request/response style protocol, there is a lot of information which can not be derived from looking at a single packet.

Thus, modern (stateful) packet filters attempt to track connections (flows) and their respective protocol states for all traffic through the packet filter.

Connection tracking within linux is implemented as a netfilter module, called ip_conntrack.o.

Before describing the connection tracking subsystem, we need to describe a couple of definitions and primitives used throughout the conntrack code.

A connection is represented within the conntrack subsystem using *struct ip_conntrack*, also called *connection tracking entry*.

Connection tracking is utilizing *conntrack tuples*, which are tuples consisting out of (srcip, srcport, dstip, dstport, l4prot). A connection is uniquely identified by two tuples: The tuple in the original direction (IP_CT_DIR_ORIGINAL) and the tuple for the reply direction (IP_CT_DIR_REPLY).

Connection tracking itself does not drop packets[3] or impose any policy. It just associates every packet with a connection tracking entry, which in turn has a particular state. All other kernel code can use this state information[4].

2.3.1 Integration of conntrack with netfilter

If the ip_conntrack.o module is registered with netfilter, it attaches to the NF_IP_PRE_ROUTING, NF_IP_POST_ROUTING, NF_IP_LOCAL_IN and NF_IP_LOCAL_OUT hooks.

Because forwarded packets are the most common case on firewalls, I will only describe how connection tracking works for forwarded packets. The two relevant hooks for forwarded packets are NF_IP_PRE_ROUTING and NF_IP_POST_ROUTING.

Every time a packet arrives at the NF_IP_PRE_ROUTING hook, connection tracking creates a conntrack tuple from the packet. It then compares this tuple to the original and reply tuples of all already-seen connections [5] to find out if this just-arrived packet belongs to any existing connection. If there is no match, a new conntrack table entry (struct ip_conntrack) is created.

Let's assume the case where we have already existing connections but are starting from scratch.

The first packet comes in, we derive the tuple from the packet headers, look up the conntrack hash table, don't find any matching entry. As a result, we create a new struct ip_conntrack. This struct ip_conntrack is filled with all necessarry data, like the original and reply tuple of the connection. How do we know the reply tuple? By inverting the source and destination parts of the original tuple.[6] Please note that this new struct ip_conntrack is **not** yet placed into the conntrack hash table.

The packet is now passed on to other callback functions which have registered with a lower priority at NF_IP_PRE_ROUTING. It then continues traversal of the network stack as usual, including all respective netfilter hooks.

If the packet survives (i.e. is not dropped by the routing code, network stack, firewall ruleset, ...), it re-appears at NF_IP_POST_ROUTING. In this case, we can now safely assume that this packet will be sent off on the outgoing interface, and thus put the connection tracking entry which we created at NF_IP_PRE_ROUTING into the conntrack hash table. This process is called *confirming the conntrack*.

The connection tracking code itself is not monolithic, but consists out of a couple of seperate modules[7]. Besides the conntrack core, there are two important kind of modules: Protocol helpers and application helpers.

[3] well, in some rare cases in combination with NAT it needs to drop. But don't tell anyone, this is secret.

[4] state information is internally represented via the *struct sk_buff.nfct* structure member of a packet.

[5] Of course this is not implemented as a linear search over all existing connections.

[6] So why do we need two tuples, if they can be derived from each other? Wait until we discuss NAT.

[7] They don't actually have to be seperate kernel modules; e.g. TCP, UDP and ICMP tracking modules are all part of the linux kernel module ip_conntrack.o

Protocol helpers implement the layer-4-protocol specific parts. They currently exist for TCP, UDP and ICMP (an experimental helper for GRE exists).

2.3.2 TCP connection tracking

As TCP is a connection oriented protocol, it is not very difficult to imagine how conntection tracking for this protocol could work. There are well-defined state transitions possible, and conntrack can decide which state transitions are valid within the TCP specification. In reality it's not all that easy, since we cannot assume that all packets that pass the packet filter actually arrive at the receiving end, ...

It is noteworthy that the standard connection tracking code does **not** do TCP sequence number and window tracking. A well-maintained patch to add this feature exists almost as long as connection tracking itself. It will be integrated with the 2.5.x kernel. The problem with window tracking is it's bad interaction with connection pickup. The TCP conntrack code is able to pick up already existing connections, e.g. in case your firewall was rebooted. However, connection pickup is conflicting with TCP window tracking: The TCP window scaling option is only transferred at connection setup time, and we don't know about it in case of pickup...

2.3.3 ICMP tracking

ICMP is not really a connection oriented protocol. So how is it possible to do connection tracking for ICMP?

The ICMP protocol can be split in two groups of messages

- ICMP error messages, which sort-of belong to a different connection ICMP error messages are associated *RELATED* to a different connection. (ICMP_DEST_UNREACH, ICMP_SOURCE_QUENCH, ICMP_TIME_EXCEEDED, ICMP_PARAMETERPROB, ICMP_REDIRECT).
- ICMP queries, which have a request-¿reply character. So what the conntrack code does, is let the request have a state of *NEW*, and the reply *ESTABLISHED*. The reply closes the connection immediately. (ICMP_ECHO, ICMP_TIMESTAMP, ICMP_INFO_REQUEST, ICMP_ADDRESS)

2.3.4 UDP connection tracking

UDP is designed as a connectionless datagram protocol. But most common protocols using UDP as layer 4 protocol have bi-directional UDP communication. Imagine a DNS query, where the client sends an UDP frame to port 53 of the nameserver, and the nameserver sends back a DNS reply packet from it's UDP port 53 to the client.

Netfilter trats this as a connection. The first packet (the DNS request) is assigned a state of *NEW*, because the packet is expected to create a new 'connection'. The dns servers' reply packet is marked as *ESTABLISHED*.

2.3.5 conntrack application helpers

More complex application protocols involving multiple connections need special support by a so-called "conntrack application helper module". Modules in the stock kernel come for FTP and IRC(DCC). Netfilter CVS currently contains patches for PPTP, H.323, Eggdrop botnet, tftp ald talk. We're still lacking a lot of protocols (e.g. SIP, SMB/CIFS) - but they are unlikely to appear until somebody really needs them and either develops them on his own or funds development.

2.3.6 Integration of connection tracking with iptables

As stated earlier, conntrack doesn't impose any policy on packets. It just determines the relation of a packet to already existing connections. To base packet filtering decision on this sate information, the iptables *state* match can be used. Every packet is within one of the following categories:

- **NEW**: packet would create a new connection, if it survives

- **ESTABLISHED**: packet is part of an already established connection (either direction)
- **RELATED**: packet is in some way related to an already established connection, e.g. ICMP errors or FTP data sessions
- **INVALID**: conntrack is unable to derive conntrack information from this packet. Please note that all multicast or broadcast packets fall in this category.

2.4 NAT Subsystem

The NAT (Network Address Translation) subsystem is probably the worst documented subsystem within the whole framework. This has two reasons: NAT is nasty and complicated. The Linux 2.4.x NAT implementation is easy to use, so nobody needs to know the nasty details.

Nonetheless, as I was traditionally concentrating most on the conntrack and NAT systems, I will give a short overview.

NAT uses almost all of the previously described subsystems:

- IP tables to specify which packets to NAT in which particular way. NAT registers a "nat" table with PREROUTING, POSTROUTING and OUTPUT chains.
- Connection tracking to associate NAT state with the connection.
- Netfilter to do the actuall packet manipulation transparent to the rest of the kernel. NAT registers with NF_IP_PRE_ROUTING, NF_IP_POST_ROUTING, NF_IP_LOCAL_IN and NF_IP_LOCAL_OUT.

The NAT implementation supports all kinds of different nat; Source NAT, Destination NAT, NAT to address/port ranges, 1:1 NAT, ...

This fundamental design principle is still frequently misunderstood:
The information about which NAT mappings apply to a certain connection is only gathered once - with the first packet of every connection.

So let's start to look at the life of a poor to-be-nat'ed packet. For ease of understanding, I have chosen to describe the most frequently used NAT scenario: Source NAT of a forwarded packet. Let's assume the packet has an original source address of 1.1.1.1, an original destination address of 2.2.2.2, and is going to be SNAT'ed to 9.9.9.9. Let's further ignore the fact that there are port numbers.

Once upon a time, our poor packet arrives at NF_IP_PRE_ROUTING, where conntrack has registered with highest priority. This means that a conntrack entry with the following two tuples is created:

```
IP_CT_DIR_ORIGINAL: 1.1.1.1 -> 2.2.2.2
IP_CT_DIR_REPLY: 2.2.2.2 -> 1.1.1.1
```

After conntrack, the packet traverses the PREROUTING chain of the "nat" IP table. Since only destination NAT happens at PREROUTING, no action occurs. After it's lengthy way through the rest of the network stack, the packet arrives at the NF_IP_POST_ROUTING hook, where it traverses the POSTROUTING chain of the "nat" table. Here it hits a SNAT rule, causing the following actions:

- Fill in a *struct ip_nat_manip*, indicating the new source address and the type of NAT (source NAT at POSTROUTING). This struct is part of the conntrack entry.
- Automatically derive the inverse NAT transormation for the reply packets: Destination NAT at PREROUTING. Fill in another *struct ip_nat_manip*.
- Alter the REPLY tuple of the conntrack entry to

  ```
  IP_CT_DIR_REPLY: 2.2.2.2 -> 9.9.9.9
  ```

- Apply the SNAT transformation to the packet

Every other packt within this connection, independent of its direction, will only execute the last step. Since all NAT information is connected with the conntrack entry, there is no need to do anything but to apply the same transormations to all packets witin the same connection.

2.5 IPv6 Firewalling with ip6tables

Yes, Linux 2.4.x comes with a usable, though incomplete system to secure your IPv6 network.
The parts ported to IPv6 are

- IP tables (called IP6 tables)
- The "filter" table
- The "mangle" table
- The userspace library (libip6tc)
- The command line tool (ip6tables)

Due to the lack of conntrack and NAT[8], only traditional, stateless packet filtering is possible. Apart from the obvious matches/targets, ip6tables can match on

- *EUI64 checker*; verifies if the MAC address of the sender is the same as in the EUI64 64 least significant bits of the source IPv6 address
- *frag6 match*, matches on IPv6 fragmentation header
- *route6 match*, matches on IPv6 routing header
- *ahesp6 match*, matches on SPIDs within AH or ESP over IPv6 packets

However, the ip6tables code doesn't seem to be used very widely (yet?). So please expect some potential remaining issues, since it is not tested as heavily as iptables.

2.6 Recent Development

Please refer to the spoken word at the presentation. Development at the time this paper was written can be quite different from development at the time the presentation is held.

3 Thanks

I'd like to thank

- *Linus Torvalds* for starting this interesting UNIX-like kernel
- *Alan Cox, David Miller, Alexey Kuznetesov, Andi Kleen* for building (one of?) the world's best TCP/IP stacks.
- *Paul "Rusty" Russell* for starting the netfilter/iptables project
- *The Netfilter Core Team* for continuing the netfilter/iptables effort
- *Astaro AG* for partially funding my current netfilter/iptables work
- *Conectiva Inc.* for partially funding parts of my past netfilter/iptables work and for inviting me to live in Brazil
- *samba.org and Kommunikationsnetz Franken e.V.* for hosting the netfilter homepage, CVS, mailing lists, ...

[8]for god's sake we don't have NAT with IPv6