PABLO NEIRA AYUSO

# Netfilter's connection tracking system

Pablo Neira Ayuso has an M.S. in computer science and has worked for several companies in the IT security industry, with a focus on open source solutions. Nowadays he is a full-time teacher and researcher at the University of Seville.

*pneira@lsi.us.es*

**FILTERING POLICIES BASED UNIQUELY** on packet header information are obsolete. These days, stateful firewalls provide advanced mechanisms to let sysadmins and security experts define more intelligent policies. This article describes the implementation details of the connection tracking system provided by the Netfilter project and also presents the required background to understand it, such as an understanding of the Netfilter framework. This article will be the perfect complement to understanding the subsystem that enables the stateful firewall available in any recent Linux kernel.

## The Netfilter Framework

The Netfilter project was founded by Paul "Rusty" Russell during the 2.3.x development series. At that time the existing firewalling tool for Linux had serious drawbacks that required a full rewrite. Rusty decided to start from scratch and create the Netfilter framework, which comprises a set of hooks over the Linux network protocol stack. With the hooks, you can register kernel modules that do some kind of network packet handling at different stages.

Iptables, the popular firewalling tool for Linux, is commonly confused with the Netfilter framework itself. This is because iptables chains and hooks have the same names. But iptables is just a brick on top of the Netfilter framework.

Fortunately, Rusty spent considerable time writing documentation [1] that comes in handy for anyone willing to understand the framework, although at some point you will surely feel the need to get your hands dirty and look at the code to go further.

### THE HOOKS AND THE CALLBACK FUNCTIONS

Netfilter inserts five hooks (Fig. 1) into the Linux networking stack to perform packet handling at different stages; these are the following:

- PREROUTING: All the packets, with no exceptions, hit this hook, which is reached before the routing decision and after all the IP header sanity checks are fulfilled. Port Address Translation (NAPT) and Redirec-

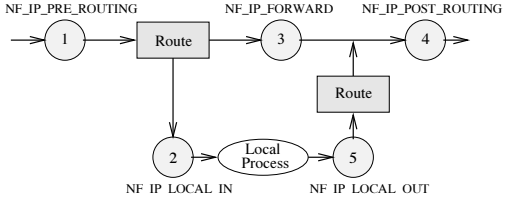


FIGURE 1: NETFILTER HOOKS

tions, that is, Destination Network Translation (DNAT), are implemented in this hook.

- LOCAL INPUT: All the packets going to the local machine reach this hook. This is the last hook in the incoming path for the local machine traffic.
- FORWARD: Packets not going to the local machine (e.g., packets going through the firewall) reach this hook.
- LOCAL OUTPUT: This is the first hook in the outgoing packet path. Packets leaving the local machine always hit this hook.
- POSTROUTING: This hook is implemented after the routing decision. Source Network Address Translation (SNAT) is registered to this hook. All the packets that leave the local machine reach this hook.

Therefore we can model three kind of traffic flows, depending on the destination:

- Traffic going through the firewall, in other words, traffic not going to the local machine. Such traffic follows the path: PREROUTING FORWARD POSTROUTING.
- Incoming traffic to the firewall, for example, traffic for the local machine. Such traffic follows the path: PREROUTING INPUT.
- Outgoing traffic from the firewall: OUTPUT POSTROUTING.

One can register a callback function to a given hook. The prototype of the callback function is defined in the structure nf_hook_ops in netfilter.h. This structure contains the information about the hook to which the callback will be registered, together with the priority. Since you can register more than one callback to a given hook, the priority indicates which callback is issued first. The register operation is done via the function nf_register_hook(...).

The callbacks can return several different values that will be interpreted by the framework in the following ways:

- ACCEPT: Lets the packet keep traveling through the stack.
- DROP: Silently discards the packet.
- QUEUE: Passes the packet to userspace via the nf_queue facility. Thus a userspace program will do the packet handling for us.
- STOLEN: Silently holds the packet until something happens, so that it temporarily does not continue to travel through the stack. This is usually used to collect defragmented IP packets.
- REPEAT: Forces the packet to reenter the hook.

In short, the framework provides a method for registering a callback function that does some kind of packet handling at any of the stages previously detailed. The return value issued will be taken by the framework that will apply the policy based on this verdict.

If at this point you consider the information provided here to be insufficient and need more background about the Linux network stack, then consult the available documentation [2] about packet travel through the Linux network stack.

## The Connection Tracking System and the Stateful Inspection

The days when packet filtering policies were based uniquely on the packet header information, such as the IP source, destination, and ports, are over. Over the years, this approach has been demonstrated to be insufficient protection against probes and denial-of-service attacks.

Fortunately, nowadays sysadmins can offer few excuses for not performing stateful filtering in their firewalls. There are open source implementations available that can be used in production environments. In the case of Linux, this feature was added during the birth of the Netfilter project. Connection tracking is another brick built on top of the Netfilter framework.

Basically, the connection tracking system stores information about the state of a connection in a memory structure that contains the source and destination IP addresses, port number pairs, protocol types, state, and timeout. With this extra information, we can define more intelligent filtering policies.

Moreover, there are some application protocols, such as FTP, TFTP, IRC, and PPTP, that have aspects that are hard to track for a firewall that follows the traditional static filtering approach. The connection tracking system defines a mechanism to track such aspects, as will be described below.

The connection tracking system does not filter the packets themselves; the default behavior always lets the packets continue their travel through the network stack, although there are a couple of very specific exceptions where packets can be dropped (e.g., under memory exhaustion). So keep in mind that the connection tracking system just tracks packets; it does not filter.

## STATES

The possible states defined for a connection are the following:

- NEW: The connection is starting. This state is reached if the packet is valid, that is, if it belongs to the valid sequence of initialization (e.g., in a TCP connection, a SYN packet is received), and if the firewall has only seen traffic in one direction (i.e., the firewall has not yet seen any reply packet).
- ESTABLISHED: The connection has been established. In other words, this state is reached when the firewall has seen two-way communication.
- RELATED: This is an expected connection. This state is further described below, in the section "Helpers and Expectations."
- INVALID: This is a special state used for packets that do not follow the expected behavior of a connection. Optionally, the sysadmin can define rules in iptables to log and drop this packet. As stated previously, connection tracking does not filter packets but, rather, provides a way to filter them.

As you have surely noticed already, by following the approach described, even stateless protocols such as UDP are stateful. And, of course, these states have nothing to do with the TCP states.

## THE BIG PICTURE

This article focuses mainly in the layer-3 independent connection tracking system implementation nf_conntrack, based on the IPv4 dependent ip_conn_track, which has been available since Linux kernel 2.6.15. Support for specific aspects of IPv4 and IPv6 are implemented in the modules nf_conntrack_ipv4 and nf_conntrack_ipv6, respectively.

Layer-4 protocol support is also implemented in separated modules. Currently, there is built-in support for TCP, UDP, ICMP, and optionally for

SCTP. These protocol handlers track the concrete aspects of a given layer-4 protocol to ensure that connections evolve correctly and that nothing evil happens.

The module nf_conntrack_ipv4 registers four callback functions (Fig. 1) in several hooks. These callbacks live in the file nf_conntrack_core.c and take as parameter the layer-3 protocol family, so basically they are the same for IPv6. The callbacks can be grouped into three families: the conntrack creation and lookup, the defragmented packets, and the helpers. The module nf_conntrack_ipv6 will not be further described in this document, since it is similar to the IPv4 variant.

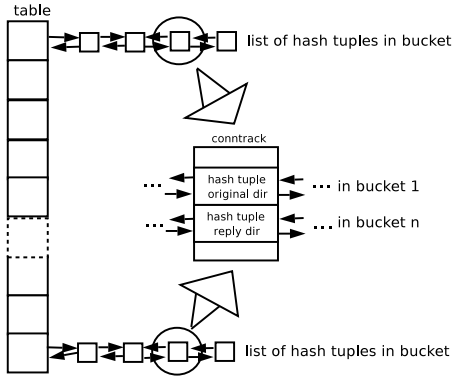


**FIGURE 2: CONNECTION TRACKING STRUCTURE**

## IMPLEMENTATION ISSUES

### BASIC STRUCTURE

The connection tracking system is an optional modular loadable subsystem, although it is always required by the NAT subsystem. It is implemented with a hash table (Fig. 2) to perform efficient lookups. Each bucket has a double-linked list of hash tuples. There are two hash tuples for every connection: one for the original direction (i.e., packets coming from the point that started the connection) and one for the reply direction (i.e., reply packets going to the point that started the connection).

A tuple represents the relevant information of a connection, IP source and IP destination, as well as layer-4 protocol information. Such tuples are embedded in a hash tuple. Both structures are defined in nf_conntrack_tuple.h.

The two hash tuples are embedded in the structure nf_conn, from this point onward referred to as *conntrack*, which is the structure that stores the state of a given connection. Therefore, a conntrack is the container of two hash tuples, and every hash tuple is the container of a tuple. This results in three layers of embedded structures.

A hash function is used to calculate the position where the hash tuple that represents the connection is supposed to be. This calculation takes as input parameters the relevant layer-3 and layer-4 protocol information. Currently, the function used is Jenkins' hash [3].

The hash calculation is augmented with a random seed to avoid the potential performance drop should some malicious user hash-bomb a given hash chain, since this can result in a very long chain of hash tuples. However, the conntrack table has a limited maximum number of conntracks; if it fills up, the evicted conntrack will be the least recently used of a hash chain. The size of the conntrack table is tunable on module load or, alternatively, at kernel boot time.

### THE CONNTRACK CREATION AND LOOKUP PROCESS

The callback nf_conntrack_in is registered in the *PREROUTING* hook. Some sanity checks are done at this stage to ensure that the packet is correct. Afterward, checks take place during the conntrack lookup process. The subsystem tries to look up a conntrack that matches with the packet received. If no conntrack is found, it will be created. This mechanism is implemented in the function resolve_normal_ct.

If the packet belongs to a new connection, the conntrack just created will

have the flag confirmed unset. The flag confirmed is set if such a conntrack is already in the hash table. This means that at this point no new conntracks are inserted. Such an insertion will happen once the packet leaves the framework successfully (i.e., when it arrives at the last hook without being dropped). The association between a packet and a conntrack is established by means of a pointer. If the pointer is null, then the packet belongs to an invalid connection. Iptables also allows us to untrack some connections. For that purpose, a dummy conntrack is used.

In conclusion, the callback nf_conntrack_confirm is registered in the LOCAL INPUT and POSTROUTING hooks. As you have already noticed, these are the last hooks in the exit path for the local and forwarded traffic, respectively. The confirmation process happens at this point: The conntrack is inserted in the hash table, the confirmed flag is set, and the associated timer is activated.

#### DEFRAGMENTED PACKET HANDLING

This work is done by the callback ipv4_conntrack_defrag, which gathers the defragmented packets. Once they are successfully received, the fragments continue their travel through the stack.

In the 2.4 kernel branch, the defragmented packets are linearized, that is, they are copied into contiguous memory. However, an optimization was introduced in kernel branch 2.6 to reduce the impact of this extra handling cost: The fragments are no longer copied into a linear space; instead, they are gathered and put in a list. Thus all handling must be fragment-aware. For example, if we need some information stored in the TCP packet header, we must first check whether the header is fragmented; if it is, then just the required information is copied to the stack. This is not actually a problem since there are available easy-to-use functions, such as skb_header_pointer, that are fragment-aware and can linearize just the portion of data required in case the packet is defragmented. Otherwise, header-checking does not incur any handling penalty.

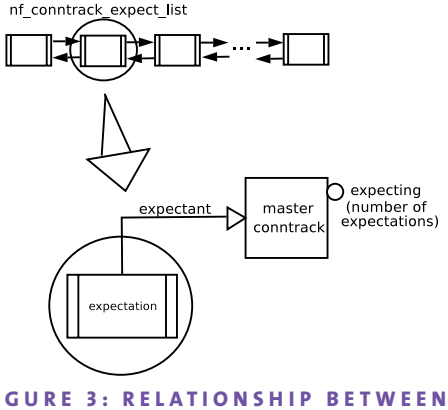


**FIGURE 3: RELATIONSHIP BETWEEN A CONNTRACK AND AN EXPECTATION**

#### HELPERS AND EXPECTATIONS

Some application-layer protocols have certain aspects that are difficult to track. For example, the File Transfer Protocol (FTP) passive mode uses port 21 for control operations to request some data from the server, but it uses TCP ports between 1024 and 65535 to receive the data requested instead of using the classical TCP port 20. This means that these two independent connections are inherently related. Therefore, the firewall requires extra information to filter this kind of protocol successfully.

The connection tracking system defines a mechanism called *helpers* that lets the system identify whether a connection is related to an existing one. To do so, it defines the concept of *expectation*. An expectation is a connection that is expected to happen in a period of time. It is defined as an nf_conntrack_expect structure in the nf_conntrack_core.h file.

The helper searches a set of patterns in the packets that contain the aspect that is hard to track. In the case of FTP, the helper looks for the PORT pattern that is sent in reply to the request to begin a passive mode connection (i.e., the PASV method). If the pattern is found, an expectation is created and is inserted in the global list of expectations (Fig. 3). Thus, the helper defines a profile of possible connections that will be expected.

An expectation has a limited lifetime. If a conntrack is created, the connection tracking system searches for matching expectations. If no matching can be found, it will look for a helper for this connection.

When the system finds a matching expectation, the new conntrack is related to the master conntrack that created such an expectation. For instance, in the case of the FTP passive mode, the conntrack that represents the traffic going to port 21 (control traffic) is the master conntrack, and the conntrack that represents the data traffic (e.g., traffic going to a high port) is related to the conntrack that represents the control traffic.

A helper is registered via nf_contrack_helper_register, which adds a structure nf_conntrack_helper to a list of helpers.

## Conclusions and Future Work

Netfilter's connection tracking system is not a piece of software stuck in time. There is considerable interesting work in progress targeted at improving the existing implementation. It is worth mentioning that during the 4th Netfilter Workshop [4], some work addressing replacing the current hash table approach with a tree of hash tables [5] was presented. The preliminary performance tests look promising.

Fortunately, the subsystem described in this document is accessible not only from the kernel side. There exists a userspace library called libnetfilter_conntrack that provides a programming interface (API) to the in-kernel connection tracking state table.

With regards to the helpers, support for Internet telephony protocols such as H.323 and VoIP are on the way. In addition, there is also some work in progress on providing the appropriate mechanisms to allow people to implement their own protocol helpers in userspace, a feature that Rusty dreamed of in the early days of the Netfilter Project.

### ACKNOWLEDGMENTS

I would like to thank Harald Welte and Patrick McHardy for spending their precious time reviewing my contributions, as well as many others. Thanks are also owed to my Ph.D. director, Rafael M. Gasca (University of Seville, Spain), and to Laurent Lefevre and the RESO/LIP laboratory (ENS Lyon, France) for the student research period of February to July 2004.

## REFERENCES

[1] Paul Russel and Harald Welte, "Netfilter Hacking How-to": http://www.netfilter.org/documentation/HOWTO/netfilter-hacking-HOWTO.txt.

[2] Miguel Rio et al., "A Map of the Networking Code in Linux Kernel 2.4.20," Technical Report DataTAG-2004-1, FP5/IST DataTAG Project, 2004.

[3] Bob Jenkins, "A Hash Function for Hash Table Lookup": http://burtleburtle.net/bob/hash/doobs.html.

[4] 4th Netfilter Workshop, October 2005: http://workshop.netfilter.org/2005/.

[5] Martin Josefsson, "Hashtrie: An Early Experiment," October 2005: http://workshop.netfilter.org/2005/presentations/martin.sxi.