# CHAPTER 9

# Netfilter

Chapter 8 discusses the IPv6 subsystem implementation. This chapter discusses the netfilter subsystem. The netfilter framework was started in 1998 by Rusty Russell, one of the most widely known Linux kernel developers, as an improvement of the older implementations of `ipchains` (Linux 2.2.x) and `ipfwadm` (Linux 2.0.x). The netfilter subsystem provides a framework that enables registering callbacks in various points (netfilter hooks) in the packet traversal in the network stack and performing various operations on packets, such as changing addresses or ports, dropping packets, logging, and more. These netfilter hooks provide the infrastructure to netfilter kernel modules that register callbacks in order to perform various tasks of the netfilter subsystem.

## Netfilter Frameworks

The netfilter subsystem provides the following functionalities, discussed in this chapter:

- Packet selection (iptables)
- Packet filtering
- Network Address Translation (NAT)
- Packet mangling (modifying the contents of packet headers before or after routing)
- Connection tracking
- Gathering network statistics

Here are some common frameworks that are based on the Linux kernel netfilter subsystem:

- **IPVS (IP Virtual Server):** A transport layer load-balancing solution (`net/netfilter/ipvs`). There is support for IPv4 IPVS from very early kernels, and support for IPVS in IPv6 is included since kernel 2.6.28. The IPv6 kernel support for IPVS was developed by Julius Volz and Vince Busam from Google. For more details, see the IPVS official website, `www.linuxvirtualserver.org`.
- **IP sets:** A framework which consists of a userspace tool called `ipset` and a kernel part (`net/netfilter/ipset`). An IP set is basically a set of IP addresses. The IP sets framework was developed by Jozsef Kadlecsik. For more details, see `http://ipset.netfilter.org`.
- **iptables:** Probably the most popular Linux firewall, **iptables** is the front end of netfilter, and it provides a management layer for netfilter: for example, adding and deleting netfilter rules, displaying statistics, adding a table, zeroing the counters of a table, and more.

There are different iptables implementations in the kernel, according to the protocol:

- **iptables** for IPv4: (`net/ipv4/netfilter/ip_tables.c`)
- **ip6tables** for IPv6: (`net/ipv6/netfilter/ip6_tables.c`)
- **arptables** for ARP: (`net/ipv4/netfilter/arp_tables.c`)
- **ebtables** for Ethernet: (`net/bridge/netfilter/ebtables.c`)

In userspace, you have the `iptables` and the `ip6tables` command-line tools, which are used to set up, maintain, and inspect the IPv4 and IPv6 tables, respectively. See `man 8 iptables` and `man 8 ip6tables`. Both `iptables` and `ip6tables` use the `setsockopt()/getsockopt()` system calls to communicate with the kernel from userspace. I should mention here two interesting ongoing netfilter projects. The `xtables2` project—being developed primarily by Jan Engelhardt, a work in progress as of this writing—uses a netlink-based interface to communicate with the kernel netfilter subsystem. See more details on the project website, `http://xtables.de`. The second project, the `nftables` project, is a new packet filtering engine that is a candidate to replace `iptables`. The `nftables` solution is based on using a virtual machine and a single unified implementation instead of the four `iptables` objects mentioned earlier (`iptables`, `ip6tables`, `arptables`, and `ebtables`). The `nftables` project was first presented in a netfilter workshop in 2008, by Patrick McHardy. The kernel infrastructure and userspace utility have been developed by Patrick McHardy and Pablo Neira Ayuso. For more details, see `http://netfilter.org/projects/nftables`, and "Nftables: a new packet filtering engine" at `http://lwn.net/Articles/324989/`.

There are a lot of netfilter modules that extend the core functionality of the core netfilter subsystem; apart from some examples, I do not describe these modules here in depth. There are a lot of information resources about these netfilter extensions from the administration perspective on the web and in various administration guides. See also the official netfilter project website: `www.netfilter.org`.

# Netfilter Hooks

There are five points in the network stack where you have netfilter hooks: you have encountered these points in previous chapters' discussions of the Rx and Tx paths in IPv4 and in IPv6. Note that the names of the hooks are common to IPv4 and IPv6:

- NF_INET_PRE_ROUTING: This hook is in the `ip_rcv()` method in IPv4, and in the `ipv6_rcv()` method in IPv6. The `ip_rcv()` method is the protocol handler of IPv4, and the `ipv6_rcv()` method is the protocol handler of IPv6. It is the first hook point that all incoming packets reach, before performing a lookup in the routing subsystem.
- NF_INET_LOCAL_IN: This hook is in the `ip_local_deliver()` method in IPv4, and in the `ip6_input()` method in IPv6. All incoming packets addressed to the local host reach this hook point after first passing via the NF_INET_PRE_ROUTING hook point and after performing a lookup in the routing subsystem.
- NF_INET_FORWARD: This hook is in the `ip_forward()` method in IPv4, and in the `ip6_forward()` method in IPv6. All forwarded packets reach this hook point after first passing via the NF_INET_PRE_ROUTING hook point and after performing a lookup in the routing subsystem.
- NF_INET_POST_ROUTING: This hook is in the `ip_output()` method in IPv4, and in the `ip6_finish_output2()` method in IPv6. Packets that are forwarded reach this hook point after passing the NF_INET_FORWARD hook point. Also packets that are created in the local machine and sent out arrive to NF_INET_POST_ROUTING after passing the NF_INET_LOCAL_OUT hook point.

- NF_INET_LOCAL_OUT: This hook is in the `__ip_local_out()` method in IPv4, and in the `__ip6_local_out()` method in IPv6. All outgoing packets that were created on the local host reach this point before reaching the NF_INET_POST_ROUTING hook point.

(`include/uapi/linux/netfilter.h`)

The NF_HOOK macro, mentioned in previous chapters, is called in some distinct points along the packet traversal in the kernel network stack; it is defined in `include/linux/netfilter.h`:

```
static inline int NF_HOOK(uint8_t pf, unsigned int hook, struct sk_buff *skb,
                          struct net_device *in, struct net_device *out,
                          int (*okfn)(struct sk_buff *))
{
    return NF_HOOK_THRESH(pf, hook, skb, in, out, okfn, INT_MIN);
}
```

The parameters of the NF_HOOK() are as follows:

- `pf`: Protocol family. NFPROTO_IPV4 for IPv4 and NFPROTO_IPV6 for IPv6.
- `hook`: One of the five netfilter hooks mentioned earlier (for example, NF_INET_PRE_ROUTING or NF_INET_LOCAL_OUT).
- `skb`: The SKB object represents the packet that is being processed.
- `in`: The input network device (`net_device` object).
- `out`: The output network device (`net_device` object). There are cases when the output device is NULL, as it is yet unknown; for example, in the `ip_rcv()` method, `net/ipv4/ip_input.c`, which is called before a routing lookup is performed, and you don't know yet which is the output device; the NF_HOOK() macro is invoked in this method with a NULL output device.
- `okfn:` A pointer to a continuation function which will be called when the hook will terminate. It gets one argument, the SKB.

The return value from a netfilter hook must be one of the following values (which are also termed *netfilter verdicts*):

- NF_DROP (0): Discard the packet silently.
- NF_ACCEPT (1): The packet continues its traversal in the kernel network stack as usual.
- NF_STOLEN (2): Do not continue traversal. The packet is processed by the hook method.
- NF_QUEUE (3): Queue the packet for user space.
- NF_REPEAT (4): The hook function should be called again.

(`include/uapi/linux/netfilter.h`)

Now that you know about the various netfilter hooks, the next section covers how netfilter hooks are registered.

## Registration of Netfilter Hooks

To register a hook callback at one of the five hook points mentioned earlier, you first define an `nf_hook_ops` object (or an array of `nf_hook_ops` objects) and then register it; the `nf_hook_ops` structure is defined in `include/linux/netfilter.h`:

```
struct nf_hook_ops {
    struct list_head list;
```

```
    /* User fills in from here down. */
    nf_hookfn      *hook;
    struct module *owner;
    u_int8_t       pf;
    unsigned int  hooknum;
    /* Hooks are ordered in ascending priority. */
    int            priority;
};
```

The following introduces some of the important members of the nf_hook_ops structure:

- hook: The hook callback you want to register. Its prototype is:

```
unsigned int nf_hookfn(unsigned int hooknum,
                       struct sk_buff *skb,
                       const struct net_device *in,
                       const struct net_device *out,
                       int (*okfn)(struct sk_buff *));
```

- pf: The protocol family (NFPROTO_IPV4 for IPv4 and NFPROTO_IPV6 for IPv6).
- hooknum: One of the five netfilter hooks mentioned earlier.
- priority: More than one hook callback can be registered on the same hook. Hook callbacks with lower priorities are called first. The nf_ip_hook_priorities enum defines possible values for IPv4 hook priorities (include/uapi/linux/netfilter_ipv4.h). See also Table 9-4 in the "Quick Reference" section at the end of this chapter.

There are two methods to register netfilter hooks:

- int nf_register_hook(struct nf_hook_ops *reg): Registers a single nf_hook_ops object.
- int nf_register_hooks(struct nf_hook_ops *reg, unsigned int n): Registers an array of *n* nf_hook_ops objects; the second parameter is the number of the elements in the array.

You will see two examples of registration of an array of nf_hook_ops objects in the next two sections. Figure 9-1 in the next section illustrates the use of priorities when registering more than one hook callback on the same hook point.

# Connection Tracking

It is not enough to filter traffic only according to the L4 and L3 headers in modern networks. You should also take into account cases when the traffic is based on sessions, such as an FTP session or a SIP session. By FTP session, I mean this sequence of events, for example: the client first creates a TCP control connection on TCP port 21, which is the default FTP port. Commands sent from the FTP client (such as listing the contents of a directory) to the server are sent on this control port. The FTP server opens a data socket on port 20, where the destination port on the client side is dynamically allocated. Traffic should be filtered according to other parameters, such as the state of a connection or timeout. This is one of the main reasons for using the Connection Tracking layer.

Connection Tracking allows the kernel to keep track of sessions. The Connection Tracking layer's primary goal is to serve as the basis of NAT. The IPv4 NAT module (net/ipv4/netfilter/iptable_nat.c) cannot be built if CONFIG_NF_CONNTRACK_IPV4 is not set. Similarly, the IPv6 NAT module (net/ipv6/netfilter/ip6table_nat.c) cannot be built if the CONFIG_NF_CONNTRACK_IPV6 is not set. However, Connection Tracking does not depend on NAT; you can run the Connection Tracking module without activating any NAT rule. The IPv4 and IPv6 NAT modules are discussed later in this chapter.

---

■ **Note** There are some userspace tools (`conntrack-tools`) for Connection Tracking administration mentioned in the "Quick Reference" section at the end of this chapter. These tools may help you to better understand the Connection Tracking layer.

---

## Connection Tracking Initialization

An array of `nf_hook_ops` objects, called `ipv4_conntrack_ops`, is defined as follows:

```
static struct nf_hook_ops ipv4_conntrack_ops[] __read_mostly = {
        {
                .hook           = ipv4_conntrack_in,
                .owner          = THIS_MODULE,
                .pf             = NFPROTO_IPV4,
                .hooknum        = NF_INET_PRE_ROUTING,
                .priority       = NF_IP_PRI_CONNTRACK,
        },
        {
                .hook           = ipv4_conntrack_local,
                .owner          = THIS_MODULE,
                .pf             = NFPROTO_IPV4,
                .hooknum        = NF_INET_LOCAL_OUT,
                .priority       = NF_IP_PRI_CONNTRACK,
        },
        {
                .hook           = ipv4_helper,
                .owner          = THIS_MODULE,
                .pf             = NFPROTO_IPV4,
                .hooknum        = NF_INET_POST_ROUTING,
                .priority       = NF_IP_PRI_CONNTRACK_HELPER,
        },
        {
                .hook           = ipv4_confirm,
                .owner          = THIS_MODULE,
                .pf             = NFPROTO_IPV4,
                .hooknum        = NF_INET_POST_ROUTING,
                .priority       = NF_IP_PRI_CONNTRACK_CONFIRM,
        },
        {
                .hook           = ipv4_helper,
                .owner          = THIS_MODULE,
                .pf             = NFPROTO_IPV4,
                .hooknum        = NF_INET_LOCAL_IN,
                .priority       = NF_IP_PRI_CONNTRACK_HELPER,
        },
        {
                .hook           = ipv4_confirm,
                .owner          = THIS_MODULE,
                .pf             = NFPROTO_IPV4,
```

```
                .hooknum        = NF_INET_LOCAL_IN,
                .priority       = NF_IP_PRI_CONNTRACK_CONFIRM,
        },
};
```

(net/ipv4/netfilter/nf_conntrack_l3proto_ipv4.c)

The two most important Connection Tracking hooks you register are the NF_INET_PRE_ROUTING hook, handled by the `ipv4_conntrack_in()` method, and the NF_INET_LOCAL_OUT hook, handled by the `ipv4_conntrack_local()` method. These two hooks have a priority of NF_IP_PRI_CONNTRACK (-200). The other hooks in the `ipv4_conntrack_ops` array have an NF_IP_PRI_CONNTRACK_HELPER (300) priority and an NF_IP_PRI_CONNTRACK_CONFIRM (INT_MAX, which is 2^31-1) priority. In netfilter hooks, a callback with a lower-priority value is executed first. (The `enum nf_ip_hook_priorities` in `include/uapi/linux/netfilter_ipv4.h` represents the possible priority values for IPv4 hooks). Both the `ipv4_conntrack_local()` method and the `ipv4_conntrack_in()` method invoke the `nf_conntrack_in()` method, passing the corresponding `hooknum` as a parameter. The `nf_conntrack_in()` method belongs to the protocol-independent NAT core, and is used both in IPv4 Connection Tracking and in IPv6 Connection Tracking; its second parameter is the protocol family, specifying whether it is IPv4 (PF_INET) or IPv6 (PF_INET6). I start the discussion with the `nf_conntrack_in()` callback. The other hook callbacks, `ipv4_confirm()` and `ipv4_help()`, are discussed later in this section.

---

■ **Note** When the kernel is built with Connection Tracking support (CONFIG_NF_CONNTRACK is set ), the Connection Tracking hook callbacks are called even if there are no iptables rules that are activated. Naturally, this has some performance cost. If the performance is very important, and you know beforehand that the device will not use the netfilter subsystem, consider building the kernel without Connection Tracking support or building Connection Tracking as a kernel module and not loading it.

---

Registration of IPv4 Connection Tracking hooks is done by calling the `nf_register_hooks()` method in the `nf_conntrack_l3proto_ipv4_init()` method (`net/ipv4/netfilter/nf_conntrack_l3proto_ipv4.c`):

```
in nf_conntrack_l3proto_ipv4_init(void) {
       . . .
       ret = nf_register_hooks(ipv4_conntrack_ops,
                               ARRAY_SIZE(ipv4_conntrack_ops))
       . . .
}
```

In Figure 9-1, you can see the Connection Tracking callbacks (`ipv4_conntrack_in()`, `ipv4_conntrack_local()`, `ipv4_helper()` and `ipv4_confirm()`), according to the hook points where they are registered.

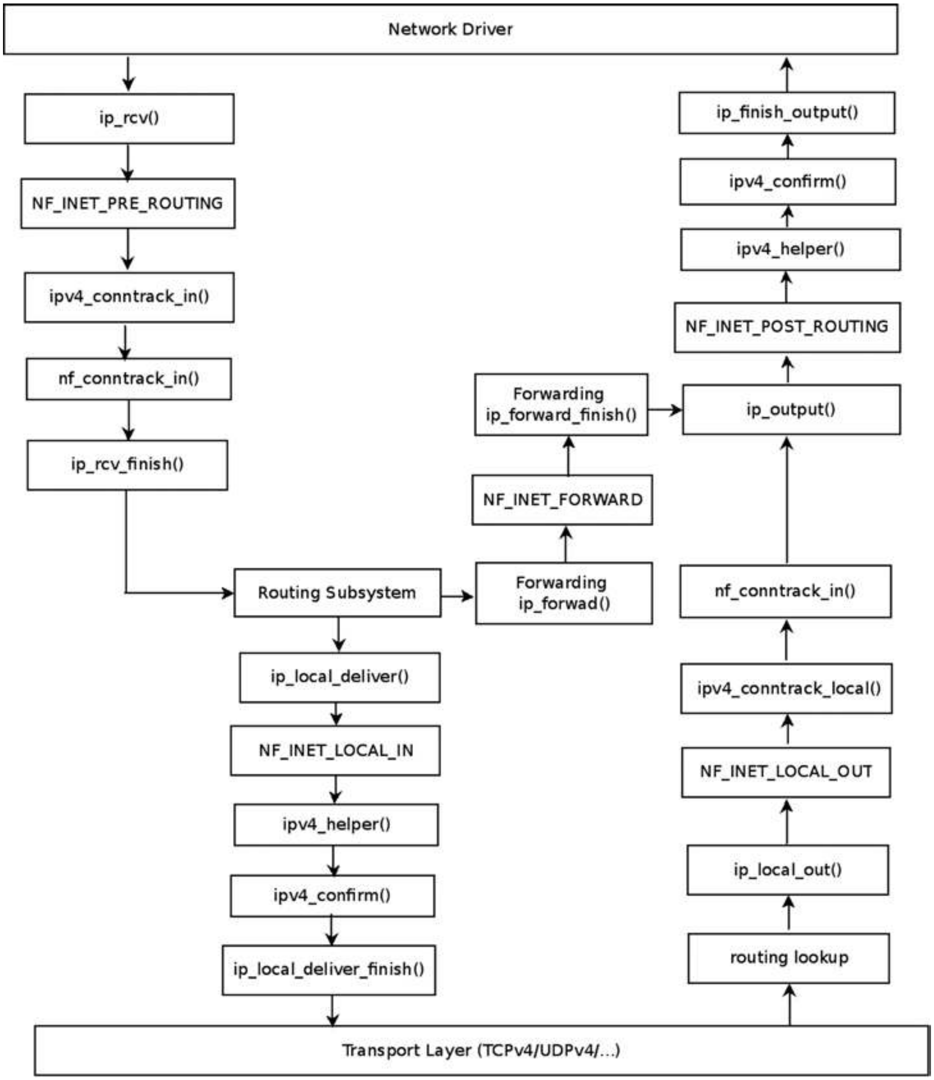


***Figure 9-1.*** *Connection Tracking hooks (IPv4)*

---

■ **Note** For the sake of simplicity, Figure 9-1 does not include more complex scenarios, such as when using IPsec or fragmentation or multicasting. It also omits the functions that are called for packets generated on the local host and sent out (like the `ip_queue_xmit()` method or the `ip_build_and_send_pkt()` method) for the sake of simplicity.

---

The basic element of Connection Tracking is the `nf_conntrack_tuple` structure:

```
struct nf_conntrack_tuple {
        struct nf_conntrack_man src;

        /* These are the parts of the tuple which are fixed. */
        struct {
                union nf_inet_addr u3;
                union {
                        /* Add other protocols here. */
                        __be16 all;

                        struct {
                                __be16 port;
                        } tcp;
                        struct {
                                __be16 port;
                        } udp;
                        struct {
                                u_int8_t type, code;
                        } icmp;
                        struct {
                                __be16 port;
                        } dccp;
                        struct {
                                __be16 port;
                        } sctp;
                        struct {
                                __be16 key;
                        } gre;
                } u;

                /* The protocol. */
                u_int8_t protonum;

                /* The direction (for tuplehash) */
                u_int8_t dir;
        } dst;
};
```

(`include/net/netfilter/nf_conntrack_tuple.h`)

The `nf_conntrack_tuple` structure represents a flow in one direction. The union inside the `dst` structure includes various protocol objects (like TCP, UDP, ICMP, and more). For each transport layer (L4) protocol, there is a Connection Tracking module, which implements the protocol-specific part. Thus, for example, you have `net/netfilter/nf_conntrack_proto_tcp.c` for the TCP protocol, `net/netfilter/nf_conntrack_proto_udp.c` for the UDP protocol, `net/netfilter/nf_conntrack_ftp.c` for the FTP protocol, and more; these modules support both IPv4 and IPv6. You will see examples of how protocol-specific implementations of Connection Tracking modules differ later in this section.

## Connection Tracking Entries

The `nf_conn` structure represents the Connection Tracking entry:

```
struct nf_conn {
        /* Usage count in here is 1 for hash table/destruct timer, 1 per skb,
           plus 1 for any connection(s) we are `master' for */
        struct nf_conntrack ct_general;

        spinlock_t lock;

        /* XXX should I move this to the tail ? - Y.K */
        /* These are my tuples; original and reply */
        struct nf_conntrack_tuple_hash tuplehash[IP_CT_DIR_MAX];

        /* Have we seen traffic both ways yet? (bitset) */
        unsigned long status;

        /* If we were expected by an expectation, this will be it */
        struct nf_conn *master;

        /* Timer function; drops refcnt when it goes off. */
        struct timer_list timeout;

      . . .

        /* Extensions */
        struct nf_ct_ext *ext;
#ifdef CONFIG_NET_NS
        struct net *ct_net;
#endif

        /* Storage reserved for other modules, must be the last member */
        union nf_conntrack_proto proto;
};
```

(include/net/netfilter/nf_conntrack.h)

The following is a description of some of the important members of the `nf_conn` structure :

- `ct_general`: A reference count.
- `tuplehash`: There are two `tuplehash` objects: `tuplehash[0]` is the original direction, and `tuplehash[1]` is the reply. They are usually referred to as `tuplehash[IP_CT_DIR_ORIGINAL]` and `tuplehash[IP_CT_DIR_REPLY]`, respectively.
- `status`: The status of the entry. When you start to track a connection entry, it is IP_CT_NEW; later on, when the connection is established, it becomes IP_CT_ESTABLISHED. See the `ip_conntrack_info` enum in `include/uapi/linux/netfilter/nf_conntrack_common.h`.

- master: An expected connection. Set by the init_conntrack() method, when an expected packet arrives (this means that the nf_ct_find_expectation() method, which is invoked by the init_conntrack() method, finds an expectation). See also the "Connection Tracking Helpers and Expectations" section later in this chapter.
- timeout: Timer of the connection entry. Each connection entry is expired after some time interval when there is no traffic. The time interval is determined according to the protocol. When allocating an nf_conn object with the __nf_conntrack_alloc() method, the timeout timer is set to be the death_by_timeout() method.

Now that you know about the nf_conn struct and some of its members, let's take a look at the nf_conntrack_in() method:

```
unsigned int nf_conntrack_in(struct net *net, u_int8_t pf, unsigned int hooknum,
                          struct sk_buff *skb)
{
        struct nf_conn *ct, *tmpl = NULL;
        enum ip_conntrack_info ctinfo;
        struct nf_conntrack_l3proto *l3proto;
        struct nf_conntrack_l4proto *l4proto;
        unsigned int *timeouts;
        unsigned int dataoff;
        u_int8_t protonum;
        int set_reply = 0;
        int ret;

        if (skb->nfct) {
                /* Previously seen (loopback or untracked)?  Ignore. */
                tmpl = (struct nf_conn *)skb->nfct;
                if (!nf_ct_is_template(tmpl)) {
                        NF_CT_STAT_INC_ATOMIC(net, ignore);
                        return NF_ACCEPT;
                }
                skb->nfct = NULL;
        }
```

First you try to find whether the network layer (L3) protocol can be tracked:

```
        l3proto = __nf_ct_l3proto_find(pf);
```

Now you try to find if the transport layer (L4) protocol can be tracked. For IPv4, it is done by the ipv4_get_l4proto() method (net/ipv4/netfilter/nf_conntrack_l3proto_ipv4):

```
        ret = l3proto->get_l4proto(skb, skb_network_offset(skb),
                           &dataoff, &protonum);
        if (ret <= 0) {
           . . .
                ret = -ret;
                goto out;
        }
```

```
l4proto = __nf_ct_l4proto_find(pf, protonum);

/* It may be an special packet, error, unclean...
 * inverse of the return code tells to the netfilter
 * core what to do with the packet. */
```

Now you check protocol-specific error conditions (see, for example, the `udp_error()` method in `net/netfilter/nf_conntrack_proto_udp.c`, which checks for malformed packets, packets with invalid checksum, and more, or the `tcp_error()` method, in `net/netfilter/nf_conntrack_proto_tcp.c`):

```
if (l4proto->error != NULL) {
        ret = l4proto->error(net, tmpl, skb, dataoff, &ctinfo,
                             pf, hooknum);
        if (ret <= 0) {
                NF_CT_STAT_INC_ATOMIC(net, error);
                NF_CT_STAT_INC_ATOMIC(net, invalid);
                ret = -ret;
                goto out;
        }
        /* ICMP[v6] protocol trackers may assign one conntrack. */
        if (skb->nfct)
                goto out;
}
```

The `resolve_normal_ct()` method, which is invoked hereafter immediately, performs the following:

- Calculates the hash of the tuple by calling the `hash_conntrack_raw()` method.
- Performs a lookup for a tuple match by calling the `__nf_conntrack_find_get()` method, passing the hash as a parameter.
- If no match is found, it creates a new `nf_conntrack_tuple_hash` object by calling the `init_conntrack()` method. This `nf_conntrack_tuple_hash` object is added to the list of unconfirmed tuplehash objects. This list is embedded in the network namespace object; the `net` structure contains a `netns_ct` object, which consists of network namespace specific Connection Tracking information. One of its members is `unconfirmed`, which is a list of unconfirmed `tuplehash` objects (see `include/net/netns/conntrack.h`). Later on, in the `__nf_conntrack_confirm()` method, it will be removed from the unconfirmed list. I discuss the `__nf_conntrack_confirm()` method later in this section.
- Each SKB has a member called `nfctinfo`, which represents the connection state (for example, it is IP_CT_NEW for new connections), and also a member called `nfct` (an instance of the `nf_conntrack struct`) which is in fact a reference counter. The `resolve_normal_ct()` method initializes both of them.

```
ct = resolve_normal_ct(net, tmpl, skb, dataoff, pf, protonum,
                       l3proto, l4proto, &set_reply, &ctinfo);
if (!ct) {
        /* Not valid part of a connection */
        NF_CT_STAT_INC_ATOMIC(net, invalid);
        ret = NF_ACCEPT;
        goto out;
}
```

```
        if (IS_ERR(ct)) {
                /* Too stressed to deal. */
                NF_CT_STAT_INC_ATOMIC(net, drop);
                ret = NF_DROP;
                goto out;
        }

        NF_CT_ASSERT(skb->nfct);
```

You now call the `nf_ct_timeout_lookup()` method to decide what timeout policy you want to apply to this flow. For example, for UDP, the timeout is 30 seconds for unidirectional connections and 180 seconds for bidirectional connections; see the definition of the `udp_timeouts` array in `net/netfilter/nf_conntrack_proto_udp.c`. For TCP, which is a much more complex protocol, there are 11 entries in `tcp_timeouts` array (`net/netfilter/nf_conntrack_proto_tcp.c`):

```
        /* Decide what timeout policy we want to apply to this flow. */
        timeouts = nf_ct_timeout_lookup(net, ct, l4proto);
```

You now call the protocol-specific `packet()` method (for example, the `udp_packet()` for UDP or the `tcp_packet()` method for TCP). The `udp_packet()` method extends the timeout according to the status of the connection by calling the `nf_ct_refresh_acct()` method. For unreplied connections (where the IPS_SEEN_REPLY_BIT flag is not set), it will be set to 30 seconds, and for replied connections, it will be set to 180. Again, in the case of TCP, the `tcp_packet()` method is much more complex, due to the TCP advanced state machine. Moreover, the `udp_packet()` method always returns a verdict of NF_ACCEPT, whereas the `tcp_packet()` method may sometimes fail:

```
        ret = l4proto->packet(ct, skb, dataoff, ctinfo, pf, hooknum, timeouts);
        if (ret <= 0) {
                 /* Invalid: inverse of the return code tells
                  * the netfilter core what to do */
                 pr_debug("nf_conntrack_in: Can't track with proto module\n");
                 nf_conntrack_put(skb->nfct);
                 skb->nfct = NULL;
                 NF_CT_STAT_INC_ATOMIC(net, invalid);
                 if (ret == -NF_DROP)
                         NF_CT_STAT_INC_ATOMIC(net, drop);
                 ret = -ret;
                 goto out;
        }

        if (set_reply && !test_and_set_bit(IPS_SEEN_REPLY_BIT, &ct->status))
                 nf_conntrack_event_cache(IPCT_REPLY, ct);
        out:
        if (tmpl) {
                 /* Special case: we have to repeat this hook, assign the
                  * template again to this packet. We assume that this packet
                  * has no conntrack assigned. This is used by nf_ct_tcp. */
                 if (ret == NF_REPEAT)
                         skb->nfct = (struct nf_conntrack *)tmpl;
                 else
                         nf_ct_put(tmpl);
        }

        return ret;
}
```

The ipv4_confirm() method, which is called in the NF_INET_POST_ROUTING hook and in the NF_INET_LOCAL_IN hook, will normally call the __nf_conntrack_confirm() method, which will remove the tuple from the unconfirmed list.

## Connection Tracking Helpers and Expectations

Some protocols have different flows for data and for control—for example, FTP, the File Transfer Protocol, and SIP, the Session Initiation Protocol, which is a VoIP protocol. Usually in these protocols, the control channel negotiates some configuration setup with the other side and agrees with it on which parameters to use for the data flow. These protocols are more difficult to handle by the netfilter subsystem, because the netfilter subsystem needs to be aware that flows are related to each other. In order to support these types of protocols, the netfilter subsystem provides the Connection Tracking Helpers, which extend the Connection Tracking basic functionality. These modules create expectations (nf_conntrack_expect objects), and these expectations tell the kernel that it should expect some traffic on a specified connection and that two connections are related. Knowing that two connections are related lets you define rules on the master connection that pertain also to the related connections. You can use a simple iptables rule based on the Connection Tracking state to accept packets whose Connection Tracking state is RELATED:

```
iptables -A INPUT -m conntrack --ctstate RELATED -j ACCEPT
```

---

■ **Note** Connections can be related not only as a result of expectation. For example, an ICMPv4 error packet such as "ICMP fragmentation needed" will be related if netfilter finds a conntrack entry that matches the tuple in the ICMP-embedded L3/L4 header. See the icmp_error_message() method for more details, net/ipv4/netfilter/nf_conntrack_proto_icmp.c.

---

The Connection Tracking Helpers are represented by the nf_conntrack_helper structure (include/net/netfilter/nf_conntrack_helper.h). They are registered and unregistered by the nf_conntrack_helper_register() method and the nf_conntrack_helper_unregister() method, respectively. Thus, for example, the nf_conntrack_helper_register() method is invoked by nf_conntrack_ftp_init() (net/netfilter/nf_conntrack_ftp.c) in order to register the FTP Connection Tracking Helpers. The Connection Tracking Helpers are kept in a hash table (nf_ct_helper_hash). The ipv4_helper() hook callback is registered in two hook points, NF_INET_POST_ROUTING and NF_INET_LOCAL_IN (see the definition of ipv4_conntrack_ops array in the "Connection Tracking Initialization" section earlier). Because of this, when the FTP packet reaches the NF_INET_POST_ROUTING callback, ip_output(), or the NF_INET_LOCAL_IN callback, ip_local_deliver(), the ipv4_helper() method is invoked, and this method eventually calls the callbacks of the registered Connection Tracking Helpers. In the case of FTP, the registered helper method is the help() method, net/netfilter/nf_conntrack_ftp.c. This method looks for FTP-specific patterns, like the "PORT" FTP command; see the invocation of the find_pattern() method in the help() method, in the following code snippet (net/netfilter/nf_conntrack_ftp.c). If there is a match, an nf_conntrack_expect object is created by calling the nf_ct_expect_init() method:

```
static int help(struct sk_buff *skb,
         unsigned int protoff,
         struct nf_conn *ct,
         enum ip_conntrack_info ctinfo)
{
       struct nf_conntrack_expect *exp;
    . . .
```

```
        for (i = 0; i < ARRAY_SIZE(search[dir]); i++) {
                found = find_pattern(fb_ptr, datalen,
                                search[dir][i].pattern,
                                search[dir][i].plen,
                                search[dir][i].skip,
                                search[dir][i].term,
                                &matchoff, &matchlen,
                                &cmd,
                                search[dir][i].getnum);
                if (found) break;
        }

        if (found == -1) {
               /* We don't usually drop packets.  After all, this is
                  connection tracking, not packet filtering.
                  However, it is necessary for accurate tracking in
                  this case. */
               nf_ct_helper_log(skb, ct, "partial matching of `%s'",
                                search[dir][i].pattern);
```

---

**Note** Normally, Connection Tracking does not drop packets. There are some cases when, due to some error or abnormal situation, packets are dropped. The following is an example of such a case: the invocation of `find_pattern()` earlier returned –1, which means that there is only a partial match; and the packet is dropped due to not finding a full pattern match.

---

```
                ret = NF_DROP;
                goto out;
        } else if (found == 0) { /* No match */
                ret = NF_ACCEPT;
                goto out_update_nl;
        }

        pr_debug("conntrack_ftp: match `%.*s' (%u bytes at %u)\n",
                 matchlen, fb_ptr + matchoff,
                 matchlen, ntohl(th->seq) + matchoff);

        exp = nf_ct_expect_alloc(ct);
    . . .
        nf_ct_expect_init(exp, NF_CT_EXPECT_CLASS_DEFAULT, cmd.l3num,
                          &ct->tuplehash[!dir].tuple.src.u3, daddr,
                          IPPROTO_TCP, NULL, &cmd.u.tcp.port);
    . . .
}
```

(net/netfilter/nf_conntrack_ftp.c)

Later on, when a new connection is created by the `init_conntrack()` method, you check whether it has expectations, and if it does, you set the IPS_EXPECTED_BIT flag and set the master of the connection (`ct->master`) to refer to the connection that created the expectation:

```
static struct nf_conntrack_tuple_hash *
init_conntrack(struct net *net, struct nf_conn *tmpl,
               const struct nf_conntrack_tuple *tuple,
               struct nf_conntrack_l3proto *l3proto,
               struct nf_conntrack_l4proto *l4proto,
               struct sk_buff *skb,
               unsigned int dataoff, u32 hash)
{
        struct nf_conn *ct;
        struct nf_conn_help *help;
        struct nf_conntrack_tuple repl_tuple;
        struct nf_conntrack_ecache *ecache;
        struct nf_conntrack_expect *exp;
        u16 zone = tmpl ? nf_ct_zone(tmpl) : NF_CT_DEFAULT_ZONE;
        struct nf_conn_timeout *timeout_ext;
        unsigned int *timeouts;

        . . .
        ct = __nf_conntrack_alloc(net, zone, tuple, &repl_tuple, GFP_ATOMIC,
                                  hash);
    . . .

        exp = nf_ct_find_expectation(net, zone, tuple);
        if (exp) {
                pr_debug("conntrack: expectation arrives ct=%p exp=%p\n",
                         ct, exp);
                /* Welcome, Mr. Bond.  We've been expecting you... */
                __set_bit(IPS_EXPECTED_BIT, &ct->status);
                ct->master = exp->master;
                if (exp->helper) {
                        help = nf_ct_helper_ext_add(ct, exp->helper,
                                                    GFP_ATOMIC);
                        if (help)
                                rcu_assign_pointer(help->helper, exp->helper);
                }
        . . .
```

Note that helpers listen on a predefined port. For example, the FTP Connection Tracking Helper listens on port 21 (see FTP_PORT definition in `include/linux/netfilter/nf_conntrack_ftp.h`). You can set a different port (or ports) in one of two ways: the first way is by a module parameter—you can override the default port value by supplying a single port or a comma-separated list of ports to the `modprobe` command:

```
modprobe nf_conntrack_ftp ports=2121
modprobe nf_conntrack_ftp ports=2022,2023,2024
```

The second way is by using the CT target:

```
iptables -A PREROUTING -t raw -p tcp --dport 8888 -j CT --helper ftp
```

Note that the CT target (net/netfilter/xt_CT.c) was added in kernel 2.6.34.

---

■ **Note** Xtables target extensions are represented by the xt_target structure and are registered by the xt_register_target() method for a single target, or by the xt_register_targets() method for an array of targets. Xtables match extensions are represented by the xt_match structure and are registered by the xt_register_match() method, or by the xt_register_matches() for an array of matches. The match extensions inspect a packet according to some criterion defined by the match extension module; thus, for example, the xt_length match module (net/netfilter/xt_length.c) inspects packets according to their length (the tot_len of the SKB in case of IPv4 packet), and the xt_connlimit module (net/netfilter/xt_connlimit.c) limits the number of parallel TCP connections per IP address.

---

This section detailed the Connection Tracking initialization. The next section deals with iptables, which is probably the most known part of the netfilter framework.

## IPTables

There are two parts to iptables. The kernel part—the core is in net/ipv4/netfilter/ip_tables.c for IPv4, and in net/ipv6/netfilter/ip6_tables.c for IPv6. And there is the userspace part, which provides a front end for accessing the kernel iptables layer (for example, adding and deleting rules with the iptables command). Each table is represented by the xt_table structure (defined in include/linux/netfilter/x_tables.h). Registration and unregistration of a table is done by the ipt_register_table() and the ipt_unregister_table() methods, respectively. These methods are implemented in net/ipv4/netfilter/ip_tables.c. In IPv6, you also use the xt_table structure for creating tables, but registration and unregistration of a table is done by the ip6t_register_table() method and the ip6t_unregister_table() method, respectively.

The network namespace object contains IPv4- and IPv6-specific objects (netns_ipv4 and netns_ipv6, respectively). The netns_ipv4 and netns_ipv6 objects, in turn, contain pointers to xt_table objects. For IPv4, in struct netns_ipv4 you have, for example, iptable_filter, iptable_mangle, nat_table, and more (include/net/netns/ipv4.h). In struct netns_ipv6 you have, for example, ip6table_filter, ip6table_mangle, ip6table_nat, and more (include/net/netns/ipv6.h). For a full list of the IPv4 and of the IPv6 network namespace netfilter tables and the corresponding kernel modules, see Tables 9-2 and 9-3 in the “Quick Reference” section at the end of this chapter.

To understand how iptables work, let’s take a look at a real example with the filter table. For the sake of simplicity, let’s assume that the filter table is the only one that is built, and also that the LOG target is supported; the only rule I am using is for logging, as you will shortly see. First, let’s take a look at the definition of the filter table:

```
#define FILTER_VALID_HOOKS ((1 << NF_INET_LOCAL_IN) | \
                            (1 << NF_INET_FORWARD) | \
                            (1 << NF_INET_LOCAL_OUT))

static const struct xt_table packet_filter = {
        .name           = "filter",
        .valid_hooks    = FILTER_VALID_HOOKS,
        .me             = THIS_MODULE,
        .af             = NFPROTO_IPV4,
        .priority       = NF_IP_PRI_FILTER,
};
```

(net/ipv4/netfilter/iptable_filter.c)

Initialization of the table is done first by calling the `xt_hook_link()` method, which sets the `iptable_filter_hook()` method as the hook callback of the `nf_hook_ops` object of the `packet_filter` table:

```
static struct nf_hook_ops *filter_ops __read_mostly;
static int __init iptable_filter_init(void)
{
     . . .
        filter_ops = xt_hook_link(&packet_filter, iptable_filter_hook);
     . . .
}
```

Then you call the `ipt_register_table()` method (note that the IPv4 `netns` object, `net->ipv4`, keeps a pointer to the filter table, `iptable_filter`):

```
static int __net_init iptable_filter_net_init(struct net *net)
{
    . . .
       net->ipv4.iptable_filter =
                ipt_register_table(net, &packet_filter, repl);
    . . .

       return PTR_RET(net->ipv4.iptable_filter);
}
```

(`net/ipv4/netfilter/iptable_filter.c`)

Note that there are three hooks in the filter table:

- NF_INET_LOCAL_IN
- NF_INET_FORWARD
- NF_INET_LOCAL_OUT

For this example, you set the following rule, using the `iptable` command line:

```
iptables -A INPUT -p udp --dport=5001 -j LOG --log-level 1
```

The meaning of this rule is that you will dump into the syslog incoming UDP packets with destination port 5001. The `log-level` modifier is the standard syslog level in the range 0 through 7; 0 is emergency and 7 is debug. Note that when running an `iptables` command, you should specify the table you want to use with the `-t` modifier; for example, `iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE` will add a rule to the NAT table. When not specifying a table name with the `-t` modifier, you use the filter table by default. So by running `iptables -A INPUT -p udp --dport=5001 -j LOG --log-level 1`, you add a rule to the filter table.

---

■ **Note** You can set targets to iptables rules; usually these can be targets from the Linux netfilter subsystems (see the earlier example for using the LOG target). You can also write your own targets and extend the iptables userspace code to support them. See “Writing Netfilter modules,” by Jan Engelhardt and Nicolas Bouliane: http://inai.de/documents/Netfilter_Modules.pdf.

---

Note that CONFIG_NETFILTER_XT_TARGET_LOG must be set in order to use the LOG target in an iptables rule, as shown in the earlier example. You can refer to the code of `net/netfilter/xt_LOG.c` as an example of an `iptables` target module.

When a UDP packet with destination port 5001 reaches the network driver and goes up to the network layer (L3), the first hook it encounters is the NF_INET_PRE_ROUTING hook; the filter table callback does not register a hook in NF_INET_PRE_ROUTING. It has only three hooks: NF_INET_LOCAL_IN, NF_INET_FORWARD, and NF_INET_LOCAL_OUT, as mentioned earlier. So you continue to the `ip_rcv_finish()` method and perform a lookup in the routing subsystem. Now there are two cases: the packet is intended to be delivered to the local host or intended to be forwarded (let's ignore cases when the packet is to be discarded). In Figure 9-2, you can see the packet traversal in both cases.

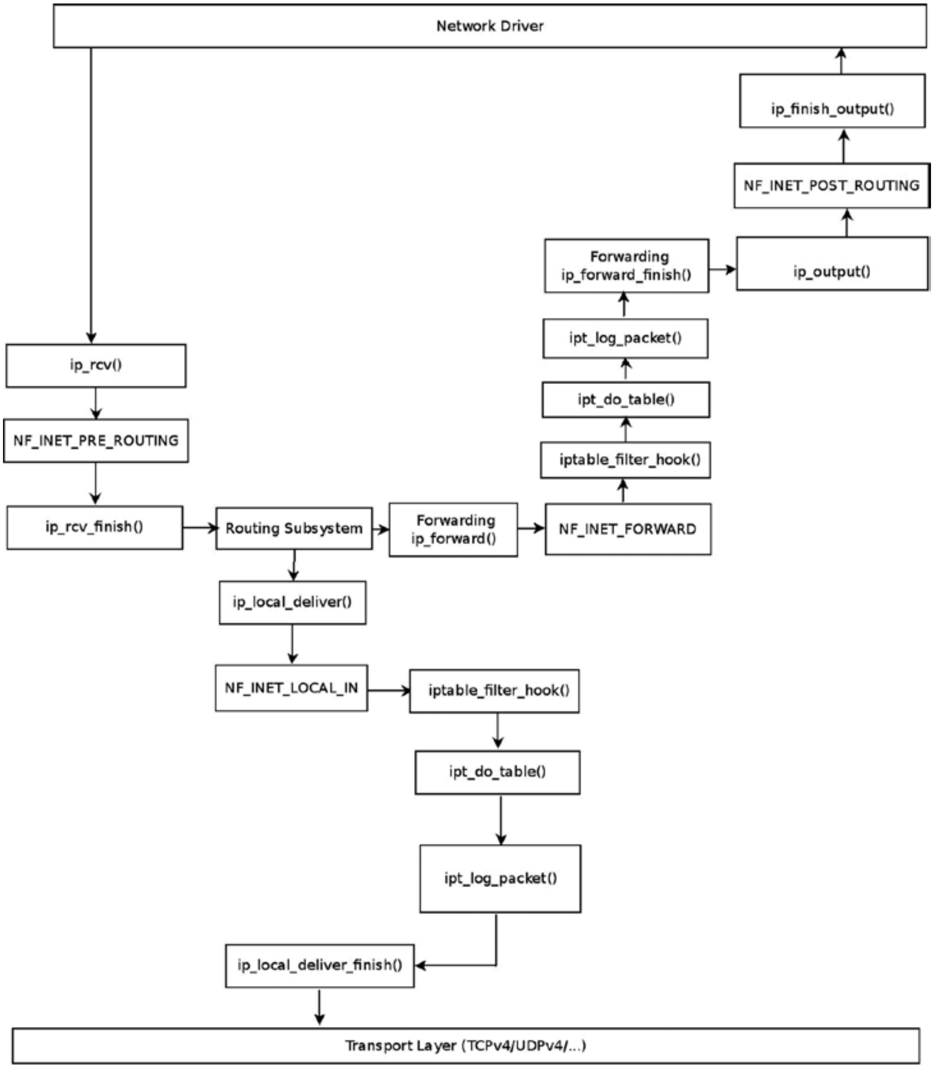


***Figure 9-2.*** *Traffic for me and Forwarded Traffic with a Filter table rule*

## Delivery to the Local Host

First you reach the ip_local_deliver() method; take a short look at this method:

```
int ip_local_deliver(struct sk_buff *skb)
{
    . . .
       return NF_HOOK(NFPROTO_IPV4, NF_INET_LOCAL_IN, skb, skb->dev, NULL,
                       ip_local_deliver_finish);
}
```

As you can see, you have the NF_INET_LOCAL_IN hook in this method, and as mentioned earlier, NF_INET_LOCAL_IN is one of the filter table hooks; so the NF_HOOK() macro will invoke the iptable_filter_hook() method. Now take a look in the iptable_filter_hook() method:

```
static unsigned int iptable_filter_hook(unsigned int hook, struct sk_buff *skb,
                                    const struct net_device *in,
                        const struct net_device *out,
                              int (*okfn)(struct sk_buff *))
{
        const struct net *net;
        . . .
        net = dev_net((in != NULL) ? in : out);
        . . .

        return ipt_do_table(skb, hook, in, out, net->ipv4.iptable_filter);
}
```

(net/ipv4/netfilter/iptable_filter.c)

The ipt_do_table() method, in fact, invokes the LOG target callback, ipt_log_packet(), which writes the packet headers into the syslog. If there were more rules, they would have been called at this point. Because there are no more rules, you continue to the ip_local_deliver_finish() method, and the packet continues its traversal to the transport layer (L4) to be handled by a corresponding socket.

## Forwarding the Packet

The second case is that after a lookup in the routing subsystem, you found that the packet is to be forwarded, so the ip_forward() method is called:

```
int ip_forward(struct sk_buff *skb)
  {
  . . .
   return NF_HOOK(NFPROTO_IPV4, NF_INET_FORWARD, skb, skb->dev,
                        rt->dst.dev, ip_forward_finish);
   . . .
```

Because the filter table has a registered hook callback in NF_INET_FORWARD, as mentioned, you again invoke the `iptable_filter_hook()` method. And consequently, as before, you again call the `ipt_do_table()` method, which will in turn again call the `ipt_log_packet()` method. You will continue to the `ip_forward_finish()` method (note that `ip_forward_finish` is the last argument of the NF_HOOK macro above, which represents the continuation method). Then call the `ip_output()` method, and because the filter table has no NF_INET_POST_ROUTING hook, you continue to the `ip_finish_output()` method.

---

■ **Note** You can filter packets according to their Connection Tracking state. The next rule will dump into syslog packets whose Connection Tracking state is ESTABLISHED:

```
iptables -A INPUT -p tcp -m conntrack --ctstate ESTABLISHED -j LOG --log-level 1
```

---

# Network Address Translation (NAT)

The Network Address Translation (NAT) module deals mostly with IP address translation, as the name implies, or port manipulation. One of the most common uses of NAT is to enable a group of hosts with a private IP address on a Local Area Network to access the Internet via some residential gateway. You can do that, for example, by setting a NAT rule. The NAT, which is installed on the gateway, can use such a rule and provide the hosts the ability to access the Web. The netfilter subsystem has NAT implementation for IPv4 and for IPv6. The IPv6 NAT implementation is mainly based on the IPv4 implementation and provides, from a user perspective, an interface similar to IPv4. IPv6 NAT support was merged in kernel 3.7. It provides some features like an easy solution to load balancing (by setting a DNAT on incoming traffic) and more. The IPv6 NAT module is in `net/ipv6/netfilter/ip6table_nat.c`. There are many types of NAT setups, and there is a lot of documentation on the Web about NAT administration. I talk about two common configurations: SNAT is source NAT, where the source IP address is changed, and DNAT is a destination NAT, where the destination IP address is changed. You can use the `-j` flag to select SNAT or DNAT. The implementation of both DNAT and SNAT is in `net/netfilter/xt_nat.c`. The next section discusses NAT initialization.

## NAT initialization

The NAT table, like the filter table in the previous section, is also an `xt_table` object. It is registered on all hook points, except for the NF_INET_FORWARD hook:

```
static const struct xt_table nf_nat_ipv4_table = {
        .name           = "nat",
        .valid_hooks    = (1 << NF_INET_PRE_ROUTING) |
                          (1 << NF_INET_POST_ROUTING) |
                          (1 << NF_INET_LOCAL_OUT) |
                          (1 << NF_INET_LOCAL_IN),
        .me             = THIS_MODULE,
        .af             = NFPROTO_IPV4,
};
```

(`net/ipv4/netfilter/iptable_nat.c`)

Registration and unregistration of the NAT table is done by calling the `ipt_register_table()` and the `ipt_unregister_table()`, respectively (`net/ipv4/netfilter/iptable_nat.c`). The network namespace (`struct net`) includes an IPv4 specific object (`netns_ipv4`), which includes a pointer to the IPv4 NAT table (`nat_table`), as mentioned in the earlier "IP tables" section. This xt_table object, which is created by the ipt_register_table() method, is assigned to this nat_table pointer. You also define an array of nf_hook_ops objects and register it:

```
 static struct nf_hook_ops nf_nat_ipv4_ops[] __read_mostly = {
        /* Before packet filtering, change destination */
        {
                .hook           = nf_nat_ipv4_in,
                .owner          = THIS_MODULE,
                .pf             = NFPROTO_IPV4,
                .hooknum        = NF_INET_PRE_ROUTING,
                .priority       = NF_IP_PRI_NAT_DST,
        },
        /* After packet filtering, change source */
        {
                .hook           = nf_nat_ipv4_out,
                .owner          = THIS_MODULE,
                .pf             = NFPROTO_IPV4,
                .hooknum        = NF_INET_POST_ROUTING,
                .priority       = NF_IP_PRI_NAT_SRC,
        },
        /* Before packet filtering, change destination */
        {
                .hook           = nf_nat_ipv4_local_fn,
                .owner          = THIS_MODULE,
                .pf             = NFPROTO_IPV4,
                .hooknum        = NF_INET_LOCAL_OUT,
                .priority       = NF_IP_PRI_NAT_DST,
        },
        /* After packet filtering, change source */
        {
                .hook           = nf_nat_ipv4_fn,
                .owner          = THIS_MODULE,
                .pf             = NFPROTO_IPV4,
                .hooknum        = NF_INET_LOCAL_IN,
                .priority       = NF_IP_PRI_NAT_SRC,
        },
};
```

Registration of the nf_nat_ipv4_ops array is done in the iptable_nat_init() method:

```
static int __init iptable_nat_init(void)
{
        int err;
        ...
        err = nf_register_hooks(nf_nat_ipv4_ops, ARRAY_SIZE(nf_nat_ipv4_ops));
        if (err < 0)
                goto err2;
        return 0;
        ...
}
```

(net/ipv4/netfilter/iptable_nat.c)

## NAT Hook Callbacks and Connection Tracking Hook Callbacks

There are some hooks on which both NAT callbacks and Connection Tracking callbacks are registered. For example, on the NF_INET_PRE_ROUTING hook (the first hook an incoming packet arrives at), there are two registered callbacks: the Connection Tracking callback, `ipv4_conntrack_in()`, and the NAT callback, `nf_nat_ipv4_in()`. The priority of the Connection Tracking callback, `ipv4_conntrack_in()`, is NF_IP_PRI_CONNTRACK (-200), and the priority of the NAT callback, `nf_nat_ipv4_in()`, is NF_IP_PRI_NAT_DST (-100). Because callbacks of the same hook with lower priorities are invoked first, the Connection Tracking `ipv4_conntrack_in()` callback, which has a priority of –200, will be invoked before the NAT `nf_nat_ipv4_in()` callback, which has a priority of –100. See Figure 9-1 for the location of the `ipv4_conntrack_in()` method and Figure 9-4 for the location of the `nf_nat_ipv4_in()`; both are in the same place, in the NF_INET_PRE_ROUTING point. The reason behind this is that NAT performs a lookup in the Connection Tracking layer, and if it does not find an entry, NAT does not perform any address translation action:

```
static unsigned int nf_nat_ipv4_fn(unsigned int hooknum,
                          struct sk_buff *skb,
                          const struct net_device *in,
                          const struct net_device *out,
                          int (*okfn)(struct sk_buff *))
{
        struct nf_conn *ct;
        . . .
        /* Don't try to NAT if this packet is not conntracked */
        if (nf_ct_is_untracked(ct))
                return NF_ACCEPT;
    . . .
}
```

(net/ipv4/netfilter/iptable_nat.c)

---

■ **Note** The `nf_nat_ipv4_fn ()` method is called from the NAT PRE_ROUTING callback, `nf_nat_ipv4_in()`.

---

On the NF_INET_POST_ROUTING hook, you have two registered Connection Tracking callbacks: the `ipv4_helper()` callback (with priority of NF_IP_PRI_CONNTRACK_HELPER, which is 300) and the `ipv4_confirm()` callback with priority of NF_IP_PRI_CONNTRACK_CONFIRM (INT_MAX, which is the highest integer value for a priority). You also have a registered NAT hook callback, `nf_nat_ipv4_out()`, with a priority of NF_IP_PRI_NAT_SRC, which is 100. As a result, when reaching the NF_INET_POST_ROUTING hook, first the NAT callback, `nf_nat_ipv4_out()`, will be called, and then the `ipv4_helper()` method will be called, and the `ipv4_confirm()` will be the last to be called. See Figure 9-4.

Let's take a look in a simple DNAT rule and see the traversal of a forwarded packet and the order in which the Connection Tracking callbacks and the NAT callbacks are called (for the sake of simplicity, assume that the filter table is not built in this kernel image). In the setup shown in Figure 9-3, the middle host (the AMD server) runs this DNAT rule:

```
iptables -t nat -A PREROUTING -j DNAT -p udp --dport 9999 --to-destination 192.168.1.8
```

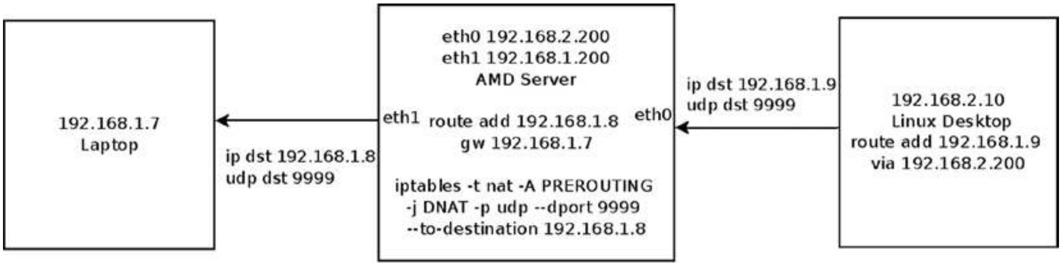


***Figure 9-3.*** *A simple setup with a DNAT rule*

The meaning of this DNAT rule is that incoming UDP packets that are sent on UDP destination port 9999 will change their destination IP address to 192.168.1.8. The right side machine (the Linux desktop) sends UDP packets to 192.168.1.9 with UDP destination port of 9999. In the AMD server, the destination IPv4 address is changed to 192.168.1.8 by the DNAT rule, and the packets are sent to the laptop on the left.

In Figure 9-4, you can see the traversal of a first UDP packet, which is sent according to the setup mentioned earlier.

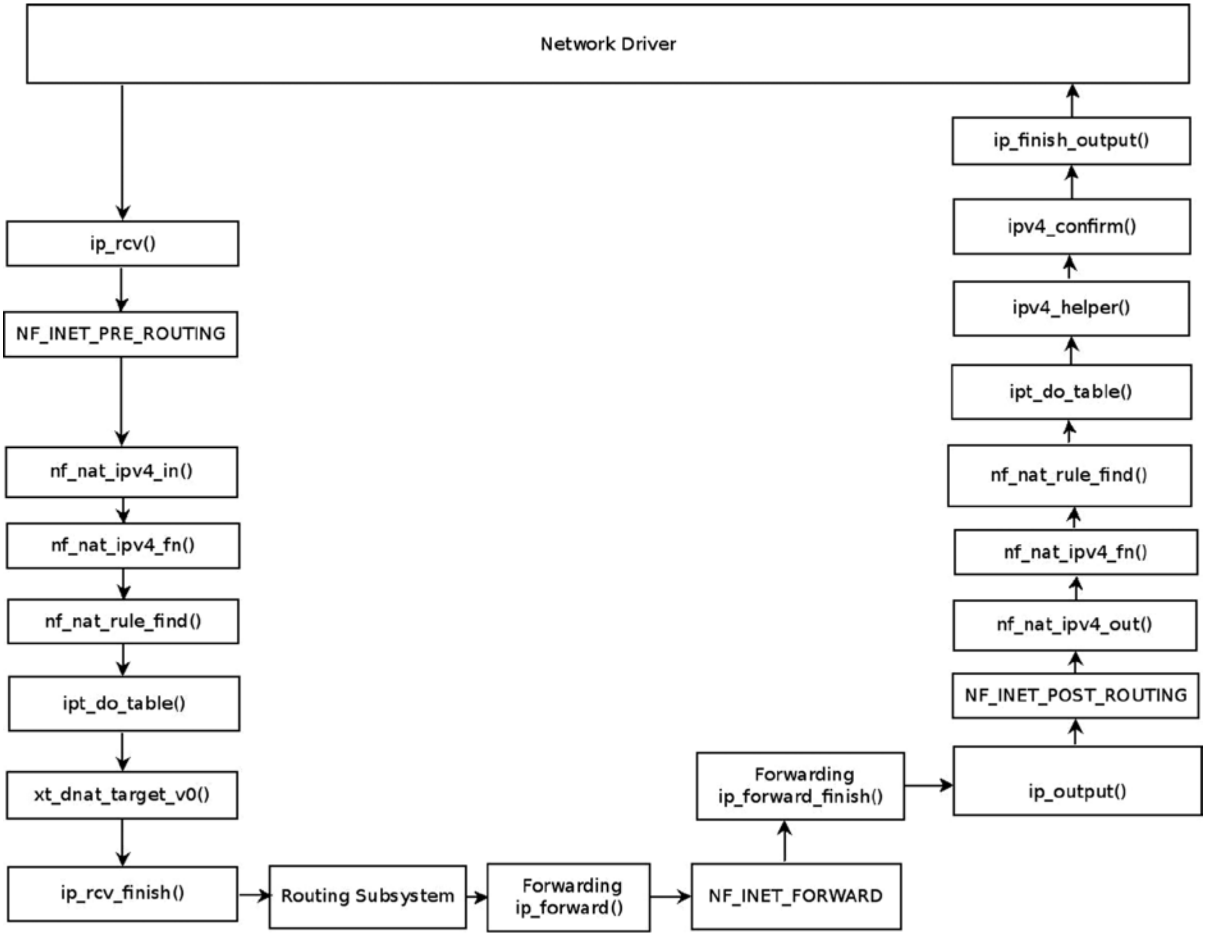


***Figure 9-4.*** *NAT and netfilter hooks*

The generic NAT module is net/netfilter/nf_nat_core.c. The basic elements of the NAT implementation are the nf_nat_l4proto structure (include/net/netfilter/nf_nat_l4proto.h) and the nf_nat_l3proto structure. In kernels prior to 3.7, you will encounter the nf_nat_protocol structure instead of these two structures, which replaced them as part of adding IPv6 NAT support. These two structures provide a protocol-independent NAT core support.

Both of these structures contain a manip_pkt() function pointer that changes the packet headers. Let's look at an example of the manip_pkt() implementation for the TCP protocol, in net/netfilter/nf_nat_proto_tcp.c:

```
static bool tcp_manip_pkt(struct sk_buff *skb,
              const struct nf_nat_l3proto *l3proto,
              unsigned int iphdroff, unsigned int hdroff,
              const struct nf_conntrack_tuple *tuple,
              enum nf_nat_manip_type maniptype)
{
        struct tcphdr *hdr;
        __be16 *portptr, newport, oldport;
        int hdrsize = 8; /* TCP connection tracking guarantees this much */

        /* this could be an inner header returned in icmp packet; in such
           cases we cannot update the checksum field since it is outside of
           the 8 bytes of transport layer headers we are guaranteed */
        if (skb->len >= hdroff + sizeof(struct tcphdr))
                hdrsize = sizeof(struct tcphdr);

        if (!skb_make_writable(skb, hdroff + hdrsize))
                return false;

        hdr = (struct tcphdr *)(skb->data + hdroff);
```

Set newport according to maniptype:

- If you need to change the source port, maniptype is NF_NAT_MANIP_SRC. So you extract the port from the tuple->src.
- If you need to change the destination port, maniptype is NF_NAT_MANIP_DST. So you extract the port from the tuple->dst:

```
        if (maniptype == NF_NAT_MANIP_SRC) {
                /* Get rid of src port */
                newport = tuple->src.u.tcp.port;
                portptr = &hdr->source;
        } else {
                /* Get rid of dst port */
                newport = tuple->dst.u.tcp.port;
                portptr = &hdr->dest;
        }
```

You are going to change the source port (when `maniptype` is NF_NAT_MANIP_SRC) or the destination port (when `maniptype` is NF_NAT_MANIP_DST) of the TCP header, so you need to recalculate the checksum. You must keep the old port for the checksum recalculation, which will be immediately done by calling the `csum_update()` method and the `inet_proto_csum_replace2()` method:

```
        oldport = *portptr;
        *portptr = newport;

        if (hdrsize < sizeof(*hdr))
                return true;
```

Recalculate the checksum:

```
        l3proto->csum_update(skb, iphdroff, &hdr->check, tuple, maniptype);
        inet_proto_csum_replace2(&hdr->check, skb, oldport, newport, 0);
        return true;
}
```

## NAT Hook Callbacks

The protocol-specific NAT module is `net/ipv4/netfilter/iptable_nat.c` for the IPv4 protocol, and `net/ipv6/netfilter/ip6table_nat.c` for the IPv6 protocol. These two NAT modules have four hooks callbacks each, shown in Table 9-1.

***Table 9-1.*** *IPv4 and IPv6 NAT Callbacks*

| Hook | Hook Callback (IPv4) | Hook Callback (IPv6) |
|---|---|---|
| NF_INET_PRE_ROUTING | nf_nat_ipv4_in | nf_nat_ipv6_in |
| NF_INET_POST_ROUTING | nf_nat_ipv4_out | nf_nat_ipv6_out |
| NF_INET_LOCAL_OUT | nf_nat_ipv4_local_fn | nf_nat_ipv6_local_fn |
| NF_INET_LOCAL_IN | nf_nat_ipv4_fn | nf_nat_ipv6_fn |

The `nf_nat_ipv4_fn()` is the most important of these methods (for IPv4). The other three methods, `nf_nat_ipv4_in()`, `nf_nat_ipv4_out()`, and `nf_nat_ipv4_local_fn()`, all invoke the `nf_nat_ipv4_fn()` method. Let's take a look at the `nf_nat_ipv4_fn()` method:

```
static unsigned int nf_nat_ipv4_fn(unsigned int hooknum,
                              struct sk_buff *skb,
                              const struct net_device *in,
                              const struct net_device *out,
                              int (*okfn)(struct sk_buff *))
{
        struct nf_conn *ct;
        enum ip_conntrack_info ctinfo;
        struct nf_conn_nat *nat;
        /* maniptype == SRC for postrouting. */
        enum nf_nat_manip_type maniptype = HOOK2MANIP(hooknum);
```

```
        /* We never see fragments: conntrack defrags on pre-routing
         * and local-out, and nf_nat_out protects post-routing.
         */
        NF_CT_ASSERT(!ip_is_fragment(ip_hdr(skb)));

        ct = nf_ct_get(skb, &ctinfo);
        /* Can't track?  It's not due to stress, or conntrack would
         * have dropped it.  Hence it's the user's responsibilty to
         * packet filter it out, or implement conntrack/NAT for that
         * protocol. 8) --RR
         */
        if (!ct)
                return NF_ACCEPT;

        /* Don't try to NAT if this packet is not conntracked */
        if (nf_ct_is_untracked(ct))
                return NF_ACCEPT;

        nat = nfct_nat(ct);
        if (!nat) {
                /* NAT module was loaded late. */
                if (nf_ct_is_confirmed(ct))
                        return NF_ACCEPT;
                nat = nf_ct_ext_add(ct, NF_CT_EXT_NAT, GFP_ATOMIC);
                if (nat == NULL) {
                        pr_debug("failed to add NAT extension\n");
                        return NF_ACCEPT;
                }
        }

        switch (ctinfo) {
        case IP_CT_RELATED:
        case IP_CT_RELATED_REPLY:
                if (ip_hdr(skb)->protocol == IPPROTO_ICMP) {
                        if (!nf_nat_icmp_reply_translation(skb, ct, ctinfo,
                                                           hooknum))
                                return NF_DROP;
                        else
                                return NF_ACCEPT;
                }
                /* Fall thru... (Only ICMPs can be IP_CT_IS_REPLY) */
        case IP_CT_NEW:
                /* Seen it before?  This can happen for loopback, retrans,
                 * or local packets.
                 */
                if (!nf_nat_initialized(ct, maniptype)) {
                        unsigned int ret;
```

The nf_nat_rule_find() method calls the ipt_do_table() method, which iterates through all the matches of an entry in a specified table, and if there is a match, calls the target callback:

```
                        ret = nf_nat_rule_find(skb, hooknum, in, out, ct);
                        if (ret != NF_ACCEPT)
                                return ret;
                } else {
                        pr_debug("Already setup manip %s for ct %p\n",
                                 maniptype == NF_NAT_MANIP_SRC ? "SRC" : "DST",
                                 ct);
                        if (nf_nat_oif_changed(hooknum, ctinfo, nat, out))
                                goto oif_changed;
                }
                break;

        default:
                /* ESTABLISHED */
                NF_CT_ASSERT(ctinfo == IP_CT_ESTABLISHED ||
                             ctinfo == IP_CT_ESTABLISHED_REPLY);
                if (nf_nat_oif_changed(hooknum, ctinfo, nat, out))
                        goto oif_changed;
        }

        return nf_nat_packet(ct, ctinfo, hooknum, skb);

oif_changed:
        nf_ct_kill_acct(ct, ctinfo, skb);
        return NF_DROP;
}
```

## Connection Tracking Extensions

Connection Tracking (CT) Extensions were added in kernel 2.6.23. The main point of Connection Tracking Extensions is to allocate only what is required—for example, if the NAT module is not loaded, the extra memory needed for NAT in the Connection Tracking layer will not be allocated. Some extensions are enabled by sysctls or even depending on certain iptables rules (for example, -m connlabel). Each Connection Tracking Extension module should define an nf_ct_ext_type object and perform registration by the nf_ct_extend_register() method (unregistration is done by the nf_ct_extend_unregister() method). Each extension should define a method to attach its Connection Tracking Extension to a connection (nf_conn) object, which should be called from the init_conntrack() method. Thus, for example, you have the nf_ct_tstamp_ext_add() method for the timestamp CT Extension and nf_ct_labels_ext_add() for the labels CT Extension. The Connection Tracking Extensions infrastructure is implemented in net/netfilter/nf_conntrack_extend.c. These are the Connection Tracking Extensions modules as of this writing (all under net/netfilter):

- nf_conntrack_timestamp.c
- nf_conntrack_timeout.c
- nf_conntrack_acct.c
- nf_conntrack_ecache.c
- nf_conntrack_labels.c
- nf_conntrack_helper.c

# Summary

This chapter described the netfilter subsystem implementation. I covered the netfilter hooks and how they are registered. I also discussed important subjects such as the Connection Tracking mechanism, iptables, and NAT. Chapter 10 deals with the IPsec subsystem and its implementation.

# Quick Reference

This section covers the top methods that are related to the topics discussed in this chapter, ordered by their context, followed by three tables and a short section about tools and libraries.

## Methods

The following is a short list of important methods of the netfilter subsystem. Some of them were mentioned in this chapter.

### struct xt_table *ipt_register_table(struct net *net, const struct xt_table *table, const struct ipt_replace *repl);

This method registers a table in the netfilter subsystem.

### void ipt_unregister_table(struct net *net, struct xt_table *table);

This method unregisters a table in the netfilter subsystem.

### int nf_register_hook(struct nf_hook_ops *reg);

This method registers a single `nf_hook_ops` object.

### int nf_register_hooks(struct nf_hook_ops *reg, unsigned int n);

This method registers an array of *n* `nf_hook_ops` objects; the second parameter is the number of the elements in the array.

### void nf_unregister_hook(struct nf_hook_ops *reg);

This method unregisters a single `nf_hook_ops` object.

### void nf_unregister_hooks(struct nf_hook_ops *reg, unsigned int n);

This method unregisters an array of *n* `nf_hook_ops` objects; the second parameter is the number of the elements in the array.

## static inline void nf_conntrack_get(struct nf_conntrack *nfct);

This method increments the reference count of the associated `nf_conntrack` object.

## static inline void nf_conntrack_put(struct nf_conntrack *nfct);

This method decrements the reference count of the associated `nf_conntrack` object. If it reaches 0, the `nf_conntrack_destroy()` method is called.

## int nf_conntrack_helper_register(struct nf_conntrack_helper *me);

This method registers an `nf_conntrack_helper` object.

## static inline struct nf_conn *resolve_normal_ct(struct net *net, struct nf_conn *tmpl, struct sk_buff *skb, unsigned int dataoff, u_int16_t l3num, u_int8_t protonum, struct nf_conntrack_l3proto *l3proto, struct nf_conntrack_l4proto *l4proto, int *set_reply, enum ip_conntrack_info *ctinfo);

This method tries to find an `nf_conntrack_tuple_hash` object according to the specified SKB by calling the `__nf_conntrack_find_get()` method, and if it does not find such an entry, it creates one by calling the `init_conntrack()` method. The `resolve_normal_ct()` method is called from the `nf_conntrack_in()` method (`net/netfilter/nf_conntrack_core.c`).

## struct nf_conntrack_tuple_hash *init_conntrack(struct net *net, struct nf_conn *tmpl, const struct nf_conntrack_tuple *tuple, struct nf_conntrack_l3proto *l3proto, struct nf_conntrack_l4proto *l4proto, struct sk_buff *skb, unsigned int dataoff, u32 hash);

This method allocates a Connection Tracking `nf_conntrack_tuple_hash` object. Invoked from the `resolve_normal_ct()` method, it tries to find an expectation for this connection by calling the `nf_ct_find_expectation()` method.

## static struct nf_conn *__nf_conntrack_alloc(struct net *net, u16 zone, const struct nf_conntrack_tuple *orig, const struct nf_conntrack_tuple *repl, gfp_t gfp, u32 hash);

This method allocates an `nf_conn` object. Sets the timeout timer of the `nf_conn` object to be the `death_by_timeout()` method.

## int xt_register_target(struct xt_target *target);

This method registers an Xtable target extension.

## void xt_unregister_target(struct xt_target *target);

This method unregisters an Xtable target extension.

## int xt_register_targets(struct xt_target *target, unsigned int n);

This method registers an array of Xtable target extensions; *n* is the number of targets.

## void xt_unregister_targets(struct xt_target *target, unsigned int n);

This method unregisters an array of Xtable target extensions; *n* is the number of targets.

## int xt_register_match(struct xt_match *target);

This method registers an Xtable match extension.

## void xt_unregister_match(struct xt_match *target);

This method unregisters an Xtable match extension.

## int xt_register_matches(struct xt_match *match, unsigned int n);

This method registers an array of Xtable match extensions; *n* is the number of matches.

## void xt_unregister_matches(struct xt_match *match, unsigned int n);

This method unregisters an array of Xtable match extensions; *n* is the number of matches.

## int nf_ct_extend_register(struct nf_ct_ext_type *type);

This method registers a Connection Tracking Extension object.

## void nf_ct_extend_unregister(struct nf_ct_ext_type *type);

This method unregisters a Connection Tracking Extension object.

## int __init iptable_nat_init(void);

This method initializes the IPv4 NAT table.

## int __init nf_conntrack_ftp_init(void);

This method initializes the Connection Tracking FTP Helper. Calls the `nf_conntrack_helper_register()` method to register the FTP helpers.

# MACRO

Let's look at the macro used in this chapter.

## NF_CT_DIRECTION(hash)

This is a macro that gets an `nf_conntrack_tuple_hash` object as a parameter and returns the direction (IP_CT_DIR_ORIGINAL, which is 0, or IP_CT_DIR_REPLY, which is 1) of the destination (`dst` object) of the associated tuple (`include/net/netfilter/nf_conntrack_tuple.h`).

## Tables

And here are the tables, showing netfilter tables in IPv4 network namespace and in IPv6 network namespace and netfilter hook priorities.

***Table 9-2.*** *IPv4 Network Namespace (`netns_ipv4`) Tables (`xt_table` Objects)*

| Linux Symbol (netns_ipv4) | Linux Module |
|---|---|
| iptable_filter | net/ipv4/netfilter/iptable_filter.c |
| iptable_mangle | net/ipv4/netfilter/iptable_mangle.c |
| iptable_raw | net/ipv4/netfilter/iptable_raw.c |
| arptable_filter | net/ipv4/netfilter/arp_tables.c |
| nat_table | net/ipv4/netfilter/iptable_nat.c |
| iptable_security | net/ipv4/netfilter/iptable_security.c (Note: CONFIG_SECURITY should be set). |

***Table 9-3.*** *IPv6 Network Namespace (`netns_ipv6`) Tables (`xt_table` Objects)*

| Linux Symbol (netns_ipv6) | Linux Module |
|---|---|
| ip6table_filter | net/ipv6/netfilter/ip6table_filter.c |
| ip6table_mangle | net/ipv6/netfilter/ip6table_mangle.c |
| ip6table_raw | net/ipv6/netfilter/ip6table_raw.c |
| ip6table_nat | net/ipv6/netfilter/ip6table_nat.c |
| ip6table_security | net/ipv6/netfilter/ip6table_security.c (Note: CONFIG_SECURITY should be set). |

***Table 9-4.*** *Netfilter Hook Priorities*

| Linux Symbol | value |
|---|---|
| NF_IP_PRI_FIRST | INT_MIN |
| NF_IP_PRI_CONNTRACK_DEFRAG | -400 |
| NF_IP_PRI_RAW | -300 |
| NF_IP_PRI_SELINUX_FIRST | -225 |
| NF_IP_PRI_CONNTRACK | -200 |
| NF_IP_PRI_MANGLE | -150 |
| NF_IP_PRI_NAT_DST | -100 |
| NF_IP_PRI_FILTER | 0 |
| NF_IP_PRI_SECURITY | 50 |
| NF_IP_PRI_NAT_SRC | 100 |
| NF_IP_PRI_SELINUX_LAST | 225 |
| NF_IP_PRI_CONNTRACK_HELPER | 300 |
| NF_IP_PRI_CONNTRACK_CONFIRM | INT_MAX |
| NF_IP_PRI_LAST | INT_MAX |

See the nf_ip_hook_priorities enum definition in include/uapi/linux/netfilter_ipv4.h.

## Tools and Libraries

The conntrack-tools consist of a userspace daemon, conntrackd, and a command line tool, conntrack. It provides a tool with which system administrators can interact with the netfilter Connection Tracking layer. See: http://conntrack-tools.netfilter.org/.

Some libraries are developed by the netfilter project and allow you to perform various userspace tasks; these libraries are prefixed with "libnetfilter"; for example, libnetfilter_conntrack, libnetfilter_log, and libnetfilter_queue. For more details, see the netfilter official website, www.netfilter.org.