CHAPTER 10

IPsec

Chapter 9 deals with the netfilter subsystem and with its kernel implementation. This chapter discusses the Internet Protocol Security (IPsec) subsystem. IPsec is a group of protocols for securing IP traffic by authenticating and encrypting each IP packet in a communication session. Most security services are provided by two major IPsec protocols: the Authentication Header (AH) protocol and the Encapsulating Security Payload (ESP) protocol. Moreover, IPsec provides protection against trying to eavesdrop and send again packets (replay attacks). IPsec is mandatory according to IPv6 specification and optional in IPv4. Nevertheless, most modern operating systems, including Linux, have support for IPsec both in IPv4 and in IPv6. The first IPsec protocols were defined in 1995 (RFCs 1825–1829). In 1998, these RFCs were deprecated by RFCs 2401–2412. Then again in 2005, these RFCs were updated by RFCs 4301–4309.

The IPsec subsystem is very complex—perhaps the most complex part of the Linux kernel network stack. Its importance is paramount when considering the growing security requirements of organizations and of private citizens. This chapter gives you a basis for delving into this complex subsystem.

General

IPsec has become a standard for most of the IP Virtual Private Network (VPN) technology in the world. That said, there are also VPNs based on different technologies, such as Secure Sockets Layer (SSL) and `pptp` (tunneling a PPP connection over the GRE protocol). Among IPsec's several modes of operation, the most important are transport mode and tunnel mode. In transport mode, only the payload of the IP packet is encrypted, whereas in tunnel mode, the entire IP packet is encrypted and inserted into a new IP packet with a new IP header. When using a VPN with IPsec, you usually work in tunnel mode, although there are cases in which you work in transport mode (L2TP/IPsec, for example).

I start with a short discussion about the Internet Key Exchange (IKE) userspace daemon and cryptography in IPsec. These are topics that are mostly not a part of the kernel networking stack but that are related to IPsec operation and are needed to get a better understanding of the kernel IPsec subsystem. I follow that with a discussion of the XFRM framework, which is the configuration and monitoring interface between the IPsec userspace part and IPsec kernel components, and explain the traversal of IPsec packets in the Tx and Rx paths. I conclude the chapter with a short section about NAT traversal in IPsec, which is an important and interesting feature, and a "Quick Reference" section. The next section begins the discussion with the IKE protocol.

IKE (Internet Key Exchange)

The most popular open source userspace Linux IPsec solutions are Openswan (and `libreswan`, which forked from Openswan), strongSwan, and racoon (of `ipsec-tools`). Racoon is part of the Kame project, which aimed to provide a free IPv6 and IPsec protocol stack implementation for variants of BSD.

To establish an IPsec connection, you need to set up a Security Association (SA). You do that with the help of the already mentioned userspace projects. An SA is defined by two parameters: a source address and a 32-bit Security Parameter Index (SPI). Both sides (called *initiator* and *responder* in IPsec terminology) should agree on parameters such as a key (or more than one key), authentication, encryption, data integrity and key exchange algorithms, and other parameters such as key lifetime (IKEv1 only). This can be done in two different ways of key distribution: by manual key exchange, which is rarely used since it is less secure, or by the IKE protocol. Openswan and strongSwan implementations provide an IKE daemon (`pluto` in Openswan and `charon` in strongSwan) that uses UDP port 500 (both source and destination) to send and receive IKE messages. Both use the XFRM Netlink interface to communicate with the native IPsec stack of the Linux kernel. The strongSwan project is the only complete open source implementation of RFC 5996, "Internet Key Exchange Protocol Version 2 (IKEv2)," whereas the Openswan project only implements a small mandatory subset.

You can use IKEv1 Aggressive Mode in Openswan and in strongSwan 5.x (for strongSwan, it should be explicitly configured, and the name of the `charon` daemon changes to be `weakSwan` in this case); but this option is regarded unsafe. IKEv1 is still used by Apple operating systems (iOS and Mac OS X) because of the built-in `racoon` legacy client. Though many implementations use IKEv1, there are many improvements and advantages when using IKEv2. I'll mention some of them very briefly: in IKEv1, more messages are needed to establish an SA than in IKEv2. IKEv1 is very complex, whereas IKEv2 is considerably simpler and more robust, mainly because each IKEv2 request message must be acknowledged by an IKEv2 response message. In IKEv1, there are no acknowledgements, but there is a backoff algorithm which, in case of packet loss, keeps trying forever. However, in IKEv1 there can be a race when the two sides perform retransmission, whereas in IKEv2 that can't happen because the responsibility for retransmission is on the initiator only. Among the other important IKEv2 features are that IKEv2 has integrated NAT traversal support, automatic narrowing of Traffic Selectors (`left|rightsubnet` on both sides don't have to match exactly, but one proposal can be a subset of the other proposal), an IKEv2 configuration payload allowing to assign virtual IPv4/IPv6 addresses and internal DNS information (replacement for IKEv1 Mode Config), and finally IKEv2 EAP authentication (replacement for the dangerous IKEv1 XAUTH protocol), which solves the problem of potentially weak PSKs by requesting a VPN server certificate and digital signature first, before the client uses a potentially weak EAP authentication algorithm (for example, EAP-MSCHAPv2).

There are two phases in IKE: the first is called Main Mode. In this stage, each side verifies the identity of the other side, and a common session key is established using the Diffie-Hellman key exchange algorithm. This mutual authentication is based on RSA or ECDSA certificates or pre-shared secrets (pre-shared key, PSKs), which are password based and assumed to be weaker. Other parameters like the Encryption algorithm and the Authentication method to be used are also negotiated. If this phase completes successfully, the two peers are said to establish an ISAKMP SA (Internet Security Association Key Management Protocol Security Association). The second phase is called Quick Mode. In this phase, both sides agree on the cryptographic algorithms to use. The IKEv2 protocol does not differentiate between phase 1 and 2 but establishes the first CHILD_SA as part of the IKE_AUTH message exchange. THE CHILD_SA_CREATE message exchange is used only to establish additional CHILD_SAs or for the periodic rekeying of the IKE and IPsec SAs. This is why IKEv1 needs nine messages to establish a single IPsec SA, whereas IKEv2 does the same in just four messages.

The next section briefly discusses cryptography in the context of IPsec (a fuller treatment of the subject would be beyond the scope of this book).

IPsec and Cryptography

There are two widely used IPsec stacks for Linux: the native Netkey stack (developed by Alexey Kuznetsov and David S. Miller) introduced with the 2.6 kernel, and the KLIPS stack, originally written for 2.0 kernel (it predates netfilter!). Netkey uses the Linux kernel Crypto API, whereas KLIPS might support more crypto hardware through Open Cryptography Framework (OCF). OCF's advantage is that it enables using asynchronous calls to encrypt/decrypt data. In the Linux kernel, most of the Crypto API performs synchronous calls. I should mention the `acrypto` kernel code, which is the asynchronous crypto layer of the Linux kernel. There are asynchronous implementations for all algorithm types. A lot of hardware crypto accelerators use the asynchronous crypto interface for crypto request offloading. That is simply because they can't block until the crypto job is done. They have to use the asynchronous API.

It is also possible to use software-implemented algorithms with the asynchronous API. For example, the `cryptd` crypto template can run arbitrary algorithms in asynchronous mode. And you can use the `pcrypt` crypto template when working in multicore environment. This template parallelizes the crypto layer by sending incoming crypto requests to a configurable set of CPUs. It also takes care of the order of the crypto requests, so it does not introduce packet reorder when used with IPsec. The use of `pcrypt` can speed up IPsec by magnitudes in some situations. The crypto layer has a user management API which is used by the `crconf` (`http://sourceforge.net/projects/crconf/`) tool to configure the crypto layer, so asynchronous crypto algorithms can be configured whenever needed. With the Linux 2.6.25 kernel, released in 2008, the XFRM framework started to offer support for the very efficient AEAD (Authenticated Encryption with Associated Data) algorithms (for example, AES-GCM), especially when the Intel AES-NI instruction set is available and data integrity comes nearly for free. Delving deeply into the details of cryptography in IPsec is beyond the scope of this book. For further information, I suggest reading the relevant chapters in *Network Security Essentials,* Fifth Edition by William Stallings (Prentice Hall, 2013).

The next section discusses the XFRM framework, which is the infrastructure of IPsec.

The XFRM Framework

IPsec is implemented by the XFRM (pronounced "transform") framework, originated in the USAGI project, which aimed at providing a production quality IPv6 and IPsec protocol stack. The term *transform* refers to an incoming packet or an outgoing packet being transformed in the kernel stack according to some IPsec rule. The XFRM framework was introduced in kernel 2.5. The XFRM infrastructure is protocol-family independent, which means that there is a generic part common to both IPv4 and IPv6, located under `net/xfrm`. Both IPv4 and IPv6 have their own implementation of ESP, AH, and IPCOMP. For example, the IPv4 ESP module is `net/ipv4/esp4.c`, and the IPv6 ESP module is `net/ipv6/esp6.c`. Apart from it, IPv4 and IPv6 implement some protocol-specific modules for supporting the XFRM infrastructure, such as `net/ipv4/xfrm4_policy.c` or `net/ipv6/xfrm6_policy.c`.

The XFRM framework supports network namespaces, which is a form of lightweight process virtualization that enables a single process or a group of processes to have their own network stack (I discuss network namespaces in Chapter 14). Each network namespace (instance of `struct net`) includes a member called `xfrm`, which is an instance of the `netns_xfrm` structure. This object includes many data structures and variables that you will encounter in this chapter, such as the hash tables of XFRM policies and the hash tables of XFRM states, `sysctl` parameters, XFRM state garbage collector, counters, and more:

```
struct netns_xfrm {
        struct hlist_head       *state_bydst;
        struct hlist_head       *state_bysrc;
        struct hlist_head       *state_byspi;
        . . .
        unsigned int            state_num;
        . . .

        struct work_struct      state_gc_work;

     . . .

        u32                     sysctl_aevent_etime;
        u32                     sysctl_aevent_rseqth;
        int                     sysctl_larval_drop;
        u32                     sysctl_acq_expires;
};

(include/net/netns/xfrm.h)
```

XFRM Initialization

In IPv4, XFRM initialization is done by calling the `xfrm_init()` method and the `xfrm4_init()` method from the `ip_rt_init()` method in `net/ipv4/route.c`. In IPv6, the `xfrm6_init()` method is invoked from the `ip6_route_init()` method for performing XFRM initialization. Communication between the userspace and the kernel is done by creating a NETLINK_XFRM netlink socket and sending and receiving netlink messages. The netlink NETLINK_XFRM kernel socket is created in the following method:

```
static int __net_init xfrm_user_net_init(struct net *net)
{
        struct sock *nlsk;
        struct netlink_kernel_cfg cfg = {
                .groups = XFRMNLGRP_MAX,
                .input  = xfrm_netlink_rcv,
        };

        nlsk = netlink_kernel_create(net, NETLINK_XFRM, &cfg);
        . . .
        return 0;
}
```

Messages sent from userspace (like XFRM_MSG_NEWPOLICY for creating a new Security Policy or XFRM_MSG_NEWSA for creating a new Security Association) are handled by the `xfrm_netlink_rcv()` method (`net/xfrm/xfrm_user.c`), which in turn calls the `xfrm_user_rcv_msg()` method (I discuss netlink sockets in Chapter 2).

The XFRM policy and the XFRM state are the fundamental data structures of the XFRM framework. I start by describing what XFRM policy is, and subsequently I describe what XFRM state is.

XFRM Policies

A Security Policy is a rule that tells IPsec whether a certain flow should be processed or whether it can bypass IPsec processing. The `xfrm_policy` structure represents an IPsec policy. A policy includes a selector (an `xfrm_selector` object). A policy is applied when its selector matches a flow. The XFRM selector consists of fields like source and destination addresses, source and destination ports, protocol, and more, which can identify a flow:

```
struct xfrm_selector {
        xfrm_address_t  daddr;
        xfrm_address_t  saddr;
        __be16  dport;
        __be16  dport_mask;
        __be16  sport;
        __be16  sport_mask;
        __u16   family;
        __u8    prefixlen_d;
        __u8    prefixlen_s;
        __u8    proto;
        int     ifindex;
        __kernel_uid32_t        user;
};
```

(include/uapi/linux/xfrm.h)

The xfrm_selector_match() method, which gets an XFRM selector, a flow, and a family (AF_INET for IPv4 or AF_INET6 for IPv6) as parameters, returns true when the specified flow matches the specified XFRM selector. Note that the xfrm_selector structure is also used in XFRM states, as you will see hereafter in this section. A Security Policy is represented by the xfrm_policy structure:

```
struct xfrm_policy {
        . . .
        struct hlist_node             bydst;
        struct hlist_node             byidx;

        /* This lock only affects elements except for entry. */
        rwlock_t                      lock;
        atomic_t                      refcnt;
        struct timer_list             timer;

        struct flow_cache_object      flo;
        atomic_t                      genid;
        u32                           priority;
        u32                           index;
        struct xfrm_mark              mark;
        struct xfrm_selector          selector;
        struct xfrm_lifetime_cfg      lft;
        struct xfrm_lifetime_cur      curlft;
        struct xfrm_policy_walk_entry walk;
        struct xfrm_policy_queue      polq;
        u8                            type;
        u8                            action;
        u8                            flags;
        u8                            xfrm_nr;
        u16                           family;
        struct xfrm_sec_ctx           *security;
        struct xfrm_tmpl              xfrm_vec[XFRM_MAX_DEPTH];
};
```

(include/net/xfrm.h)

The following description covers the important members of the xfrm_policy structure:

- refcnt: The XFRM policy reference counter; initialized to 1 in the xfrm_policy_alloc( ) method, incremented by the xfrm_pol_hold() method, and decremented by the xfrm_pol_put() method.
- timer: Per-policy timer; the timer callback is set to be xfrm_policy_timer() in the xfrm_policy_alloc() method. The xfrm_policy_timer() method handles policy expiration: it is responsible for deleting a policy when it is expired by calling the xfrm_policy_delete() method, and sending an event (XFRM_MSG_POLEXPIRE) to all registered Key Managers by calling the km_policy_expired() method.
- lft: The XFRM policy lifetime (xfrm_lifetime_cfg object). Every XFRM policy has a lifetime, which is a time interval (expressed as a time or byte count).

You can set XFRM policy lifetime values with the `ip` command and the `limit` parameter—for example:

```
ip xfrm policy add src 172.16.2.0/24 dst 172.16.1.0/24 limit byte-soft 6000 ...
```

- sets the `soft_byte_limit` of the XFRM policy lifetime (`lft`) to be 6000; see `man 8 ip xfrm`.

You can display the lifetime (`lft`) of an XFRM policy by inspecting the lifetime configuration entry when running `ip -stat xfrm policy show`.

- `curlft`: The XFRM policy current lifetime, which reflects the current status of the policy in context of lifetime. The `curlft` is an `xfrm_lifetime_cur` object. It consists of four members (all of them are fields of 64 bits, unsigned):
  - `bytes`: The number of bytes which were processed by the IPsec subsystem, incremented in the Tx path by the `xfrm_output_one()` method and in the Rx path by the `xfrm_input()` method.
  - `packets`: The number of packets that were processed by the IPsec subsystem, incremented in the Tx path by the `xfrm_output_one()` method, and in the Rx path by the `xfrm_input()` method.
  - `add_time`: The timestamp of adding the policy, initialized when adding a policy, in the `xfrm_policy_insert()` method and in the `xfrm_sk_policy_insert()` method.
  - `use_time`: The timestamp of last access to the policy. The `use_time` timestamp is updated, for example, in the `xfrm_lookup()` method or in the `__xfrm_policy_check()` method. Initialized to 0 when adding the XFRM policy, in the `xfrm_policy_insert()` method and in the `xfrm_sk_policy_insert()` method.

■ **Note** You can display the current lifetime (`curlft`) object of an XFRM policy by inspecting the lifetime current entry when running `ip -stat xfrm policy show`.

- `polq`: A queue to hold packets that are sent while there are still no XFRM states associated with the policy. As a default, such packets are discarded by calling the `make_blackhole()` method. When setting the `xfrm_larval_drop sysctl` entry to 0 (`/proc/sys/net/core/xfrm_larval_drop`), these packets are kept in a queue (`polq.hold_queue`) of SKBs; up to 100 packets (XFRM_MAX_QUEUE_LEN) can be kept in this queue. This is done by creating a dummy XFRM bundle, by the `xfrm_create_dummy_bundle()` method (see more in the "XFRM lookup" section later in this chapter). By default, the `xfrm_larval_drop sysctl` entry is set to 1 (see the `__xfrm_sysctl_init()` method in `net/xfrm/xfrm_sysctl.c`).
- `type`: Usually the type is XFRM_POLICY_TYPE_MAIN (0). When the kernel has support for subpolicy (CONFIG_XFRM_SUB_POLICY is set), two policies can be applied to the same packet, and you can use the XFRM_POLICY_TYPE_SUB (1) type. Policy that lives a shorter time in kernel should be a subpolicy. This feature is usually needed only for developers/debugging and for mobile IPv6, because you might apply one policy for IPsec and one for mobile IPv6. The IPsec policy is usually the main policy with a longer lifetime than the mobile IPv6 (sub) policy.
- `action`: Can have one of these two values:
  - XFRM_POLICY_ALLOW (0): Permit the traffic.
  - XFRM_POLICY_BLOCK(1): Disallow the traffic (for example, when using `type=reject` or `type=drop` in `/etc/ipsec.conf`).

- `xfrm_nr`: Number of templates associated with the policy—can be up to six templates (XFRM_MAX_DEPTH). The `xfrm_tmpl` structure is an intermediate structure between the XFRM state and the XFRM policy. It is initialized in the `copy_templates()` method, `net/xfrm/xfrm_user.c`.
- `family`: IPv4 or IPv6.
- `security`: A security context (`xfrm_sec_ctx` object) that allows the XFRM subsystem to restrict the sockets that can send or receive packets via Security Associations (XFRM states). For more details, see http://lwn.net/Articles/156604/.
- `xfrm_vec`: An array of XFRM templates (`xfrm_tmpl` objects).

The kernel stores the IPsec Security Policies in the Security Policy Database (SPD). Management of the SPD is done by sending messages from a userspace socket. For example:

- Adding an XFRM policy (XFRM_MSG_NEWPOLICY) is handled by the `xfrm_add_policy()` method.
- Deleting an XFRM policy (XFRM_MSG_DELPOLICY) is handled by the `xfrm_get_policy()` method.
- Displaying the SPD (XFRM_MSG_GETPOLICY) is handled by the `xfrm_dump_policy()` method.
- Flushing the SPD (XFRM_MSG_FLUSHPOLICY) is handled by the `xfrm_flush_policy()` method.

The next section describes what XFRM state is.

XFRM States (Security Associations)

The `xfrm_state` structure represents an IPsec Security Association (SA) (`include/net/xfrm.h`). It represents unidirectional traffic and includes information such as cryptographic keys, flags, request id, statistics, replay parameters, and more. You add XFRM states by sending a request (XFRM_MSG_NEWSA) from a userspace socket; it is handled in the kernel by the `xfrm_state_add()` method (`net/xfrm/xfrm_user.c`). Likewise, you delete a state by sending an XFRM_MSG_DELSA message, and it is handled in the kernel by the `xfrm_del_sa()` method:

```
struct xfrm_state {
        . . .
        union {
                struct hlist_node       gclist;
                struct hlist_node       bydst;
        };
        struct hlist_node       bysrc;
        struct hlist_node       byspi;

        atomic_t                refcnt;
        spinlock_t              lock;

        struct xfrm_id          id;
        struct xfrm_selector    sel;
        struct xfrm_mark        mark;
        u32                     tfcpad;
```

```
        u32                     genid;

        /* Key manager bits */
        struct xfrm_state_walk  km;

        /* Parameters of this state. */
        struct {
                u32             reqid;
                u8              mode;
                u8              replay_window;
                u8              aalgo, ealgo, calgo;
                u8              flags;
                u16             family;
                xfrm_address_t  saddr;
                int             header_len;
                int             trailer_len;
        } props;

        struct xfrm_lifetime_cfg lft;

        /* Data for transformer */
        struct xfrm_algo_auth   *aalg;
        struct xfrm_algo        *ealg;
        struct xfrm_algo        *calg;
        struct xfrm_algo_aead   *aead;

        /* Data for encapsulator */
        struct xfrm_encap_tmpl  *encap;

        /* Data for care-of address */
        xfrm_address_t  *coaddr;

        /* IPComp needs an IPIP tunnel for handling uncompressed packets */
        struct xfrm_state       *tunnel;

        /* If a tunnel, number of users + 1 */
        atomic_t                tunnel_users;

        /* State for replay detection */
        struct xfrm_replay_state replay;
        struct xfrm_replay_state_esn *replay_esn;

        /* Replay detection state at the time we sent the last notification */
        struct xfrm_replay_state preplay;
        struct xfrm_replay_state_esn *preplay_esn;

        /* The functions for replay detection. */
        struct xfrm_replay      *reply;
```

```
        /* internal flag that only holds state for delayed aevent at the
         * moment
        */
        u32                     xflags;

        /* Replay detection notification settings */
        u32                     replay_maxage;
        u32                     replay_maxdiff;

        /* Replay detection notification timer */
        struct timer_list       rtimer;

        /* Statistics */
        struct xfrm_stats       stats;

        struct xfrm_lifetime_cur curlft;
        struct tasklet_hrtimer  mtimer;

        /* used to fix curlft->add_time when changing date */
        long            saved_tmo;

        /* Last used time */
        unsigned long           lastused;

        /* Reference to data common to all the instances of this
         * transformer. */
        const struct xfrm_type  *type;
        struct xfrm_mode        *inner_mode;
        struct xfrm_mode        *inner_mode_iaf;
        struct xfrm_mode        *outer_mode;

        /* Security context */
        struct xfrm_sec_ctx     *security;

        /* Private data of this transformer, format is opaque,
         * interpreted by xfrm_type methods. */
        void                    *data;
};
```

(include/net/xfrm.h)

The following description details some of the important members of the `xfrm_state` structure:

- `refcnt`: A reference counter, incremented by the `xfrm_state_hold()` method and decremented by the `__xfrm_state_put()` method or by the `xfrm_state_put()` method (the latter also releases the XFRM state by calling the `__xfrm_state_destroy()` method when the reference counter reaches 0).
- `id`: The `id` (`xfrm_id` object) consists of three fields, which uniquely define it: destination address, spi, and security protocol (AH, ESP, or IPCOMP).

- `props`: The properties of the XFRM state. For example:
  - `mode`: Can be one of five modes (for example, XFRM_MODE_TRANSPORT for transport mode or XFRM_MODE_TUNNEL for tunnel mode; see `include/uapi/linux/xfrm.h`).
  - `flag`: For example, XFRM_STATE_ICMP. These flags are available in `include/uapi/linux/xfrm.h`. These flags can be set from userspace, for example, with the `ip` command and the `flag` option: `ip xfrm add state flag icmp ...`
  - `family`: IPv4 of IPv6.
  - `saddr`: The source address of the XFRM state.
  - `lft:` The XFRM state lifetime (`xfrm_lifetime_cfg` object).
  - `stats`: An `xfrm_stats` object, representing XFRM state statistics. You can display the XFRM state statistics by `ip -stat xfrm show`.

The kernel stores the IPsec Security Associations in the Security Associations Database (SAD). The `xfrm_state` objects are stored in three hash tables in `netns_xfrm` (the XFRM namespace, discussed earlier): `state_bydst`, `state_bysrc`, `state_byspi`. The keys to these tables are computed by the `xfrm_dst_hash()`, `xfrm_src_hash()`, and `xfrm_spi_hash()` methods, respectively. When an `xfrm_state` object is added, it is inserted into these three hash tables. If the value of the spi is 0 (the value 0 is not normally to be used for spi—I will shortly mention when it is 0), the `xfrm_state` object is not added to the `state_byspi` hash table (see the `__xfrm_state_insert()` method in `net/xfrm/xfrm_state.c`).

■ **Note** An spi with value of 0 is only used for acquire states. The kernel sends an acquire message to the key manager and adds a temporary acquire state with spi 0 if traffic matches a policy, but the state is not yet resolved. The kernel does not bother to send a further acquire as long as the acquire state exists; the lifetime can be configured at `net->xfrm.sysctl_acq_expires`. If the state gets resolved, this acquire state is replaced by the actual state.

Lookup in the SAD can be done by the following:

- `xfrm_state_lookup()` method: In the `state_byspi` hash table.
- `xfrm_state_lookup_byaddr()` method: In the `state_bysrc` hash table.
- `xfrm_state_find()` method: In the `state_bydst` hash table.

The ESP protocol is the most commonly used IPsec protocol; it supports both encryption and authentication. The next section discusses the IPv4 ESP implementation.

ESP Implementation (IPv4)

The ESP protocol is specified in RFC 4303; it supports both encryption and authentication. Though it also supports encryption-only and authentication-only modes, it is usually used with both encryption and authentication because it is safer. I should also mention here the new Authenticated Encryption (AEAD) methods like AES-GCM, which can do the encryption and data integrity computations in a single pass and can be highly parallelized on multiple cores, so that with the Intel AES-NI instruction set, an IPsec throughput of several Gbit/s can be achieved. The ESP protocol

supports both tunnel mode and transport mode; the protocol identifier is 50 (IPPROTO_ESP). The ESP adds a new header and a trailer to each packet. According to the ESP format, illustrated in Figure 10-1, there are the following fields:

- *SPI:* A 32-bit Security Parameter Index. Together with the source address, it identities an SA.
- *Sequence Number:* 32 bits, incremented by 1 for each transmitted packet in order to protect against replay attacks.
- *Payload Data:* A variable size encrypted data block.
- *Padding:* Padding for the encrypted data block in order to satisfy alignment requirements (0–255 bytes).
- *Pad Length:* The size of padding in bytes (1 byte).
- *Next Header:* The type of the next header (1 byte).
- *Authentication Data:* The Integrity Check Value (ICV).

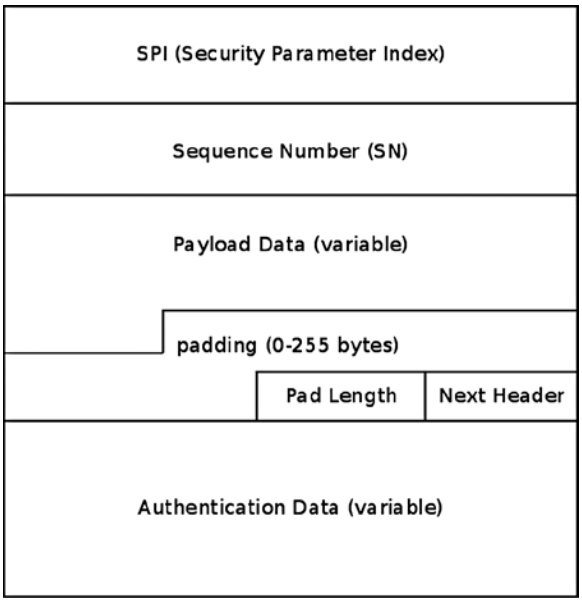

Figure 10-1. *ESP format*

The next section discusses IPv4 ESP initialization.

IPv4 ESP Initialization

We first define an esp_type (xfrm_type object) and esp4_protocol (net_protocol object) and register them thus:

```
static const struct xfrm_type esp_type =
{
        .description    = "ESP4",
        .owner          = THIS_MODULE,
        .proto          = IPPROTO_ESP,
        .flags          = XFRM_TYPE_REPLAY_PROT,
        .init_state     = esp_init_state,
        .destructor     = esp_destroy,
        .get_mtu        = esp4_get_mtu,
        .input          = esp_input,
        .output         = esp_output
};

static const struct net_protocol esp4_protocol = {
        .handler        =       xfrm4_rcv,
        .err_handler    =       esp4_err,
        .no_policy      =       1,
        .netns_ok       =       1,
};

static int __init esp4_init(void)
{
```

Each protocol family has an instance of an xfrm_state_afinfo object, which includes protocol-family specific state methods; thus there is xfrm4_state_afinfo for IPv4 (net/ipv4/xfrm4_state.c) and xfrm6_state_afinfo for IPv6. This object includes an array of xfrm_type objects called type_map. Registering XFRM type by calling the xfrm_register_type() method will set the specified xfrm_type as an element in this array:

```
        if (xfrm_register_type(&esp_type, AF_INET) < 0) {
                pr_info("%s: can't add xfrm type\n", __func__);
                return -EAGAIN;
        }
```

Registering the IPv4 ESP protocol is done like registering any other IPv4 protocol, by calling the inet_add_protocol() method. Note that the protocol handler used by IPv4 ESP, namely the xfrm4_rcv() method, is also used by the IPv4 AH protocol (net/ipv4/ah4.c) and by the IPv4 IPCOMP (IP Payload Compression Protocol) protocol (net/ipv4/ipcomp.c).

```
        if (inet_add_protocol(&esp4_protocol, IPPROTO_ESP) < 0) {
                pr_info("%s: can't add protocol\n", __func__);
                xfrm_unregister_type(&esp_type, AF_INET);
                return -EAGAIN;
        }
        return 0;
}
```

(net/ipv4/esp4.c)

Receiving an IPsec Packet (Transport Mode)

Suppose you work in transport mode in IPv4, and you receive an ESP packet that is destined to the local host. ESP in transport mode does not encrypt the IP header, only the IP payload. Figure 10-2 shows the traversal of an incoming IPv4 ESP packet, and its stages are described in this section. We will pass all the usual stages of local delivery, starting with the `ip_rcv()` method, and we will reach the `ip_local_deliver_finish()` method. Because the value of the protocol field in the IPv4 header is ESP (50), we invoke its handler, which is the `xfrm4_rcv()` method, as you saw earlier. The `xfrm4_rcv()` method further calls the generic `xfrm_input()` method, which performs a lookup in the SAD by calling the `xfrm_state_lookup()` method. If the lookup fails, the packet is dropped. In case of a lookup hit, the `input` callback method of the corresponding IPsec protocol is invoked:

```
int xfrm_input(struct sk_buff *skb, int nexthdr, __be32 spi, int encap_type)
{
        struct xfrm_state *x;
        do {
                . . .
```

Figure 10-2. *Receiving IPv4 ESP packet, local delivery, transport mode. Note: The figure describes an IPv4 ESP packet. For IPv4 AH packets, the ah_input() method is invoked instead of the esp_input( ) method; likewise, for IPv4 IPCOMP packets, the ipcomp_input() method is invoked instead of the esp_input( ) method*

Perform a lookup in the `state_byspi` hash table:

```
x = xfrm_state_lookup(net, skb->mark, daddr, spi, nexthdr, family);
```

Drop the packet silently if the lookup failed:

```
if (x == NULL) {
        XFRM_INC_STATS(net, LINUX_MIB_XFRMINNOSTATES);
        xfrm_audit_state_notfound(skb, family, spi, seq);
        goto drop;
}
```

In this case, of IPv4 ESP incoming traffic, the XFRM type associated with the state (`x->type`) is the ESP XFRM Type (`esp_type`); its `input` callback was set to `esp_input()`, as mentioned earlier in the "IPv4 ESP initialization" section.

By calling `x->type->input()`, in the following line the `esp_input()` method is invoked; this method returns the protocol number of the original packet, before it was encrypted by ESP:

```
nexthdr = x->type->input(x, skb);
. . .
```

The original protocol number is kept in the control buffer (`cb`) of the SKB by using the XFRM_MODE_SKB_CB macro; it will be used later for modifying the IPv4 header of the packet, as you will see:

```
XFRM_MODE_SKB_CB(skb)->protocol = nexthdr;
```

After the `esp_input()` method terminates, the `xfrm4_transport_finish()` method is invoked. This method modifies various fields of the IPv4 header. Take a look at the `xfrm4_transport_finish()` method:

```
int xfrm4_transport_finish(struct sk_buff *skb, int async)
{
        struct iphdr *iph = ip_hdr(skb);
```

The protocol of the IPv4 header (`iph->protocol`) is 50 (ESP) at this point; you should set it to be the protocol number of the original packet (before it was encrypted by ESP) so that it will be processed by L4 sockets. The protocol number of the original packet was kept in `XFRM_MODE_SKB_CB(skb)->protocol`, as you saw earlier in this section:

```
        iph->protocol = XFRM_MODE_SKB_CB(skb)->protocol;

        . . .
        __skb_push(skb, skb->data - skb_network_header(skb));
        iph->tot_len = htons(skb->len);
```

Recalculate the checksum, since the IPv4 header was modified:

```
        ip_send_check(iph);
```

Invoke any netfilter NF_INET_PRE_ROUTING hook callback and then call the `xfrm4_rcv_encap_finish()` method:

```
        NF_HOOK(NFPROTO_IPV4, NF_INET_PRE_ROUTING, skb, skb->dev, NULL,
                xfrm4_rcv_encap_finish);
        return 0;
}
```

The `xfrm4_rcv_encap_finish()` method calls the `ip_local_deliver()` method. Now the value of the `protocol` member in the IPv4 header is the original transport protocol (UDPv4, TCPv4, and so on), so from now on you proceed in the usual packet traversal, and the packet is passed to the transport layer (L4).

Sending an IPsec Packet (Transport Mode)

Figure 10-3 shows the Tx path of an outgoing packet sent via IPv4 ESP in transport mode. The first step after performing a lookup in the routing subsystem (by calling the `ip_route_output_flow()` method), is to perform a lookup for an XFRM policy, which can be applied on this flow. You do that by calling the `xfrm_lookup()` method (I discuss the internals of this method later in this section). If there is a lookup hit, continue to the `ip_local_out()` method, and then, after calling several methods as you can see in Figure 10-3, you eventually reach the `esp_output()` method, which encrypts the packet and then sends it out by calling the `ip_output()` method.

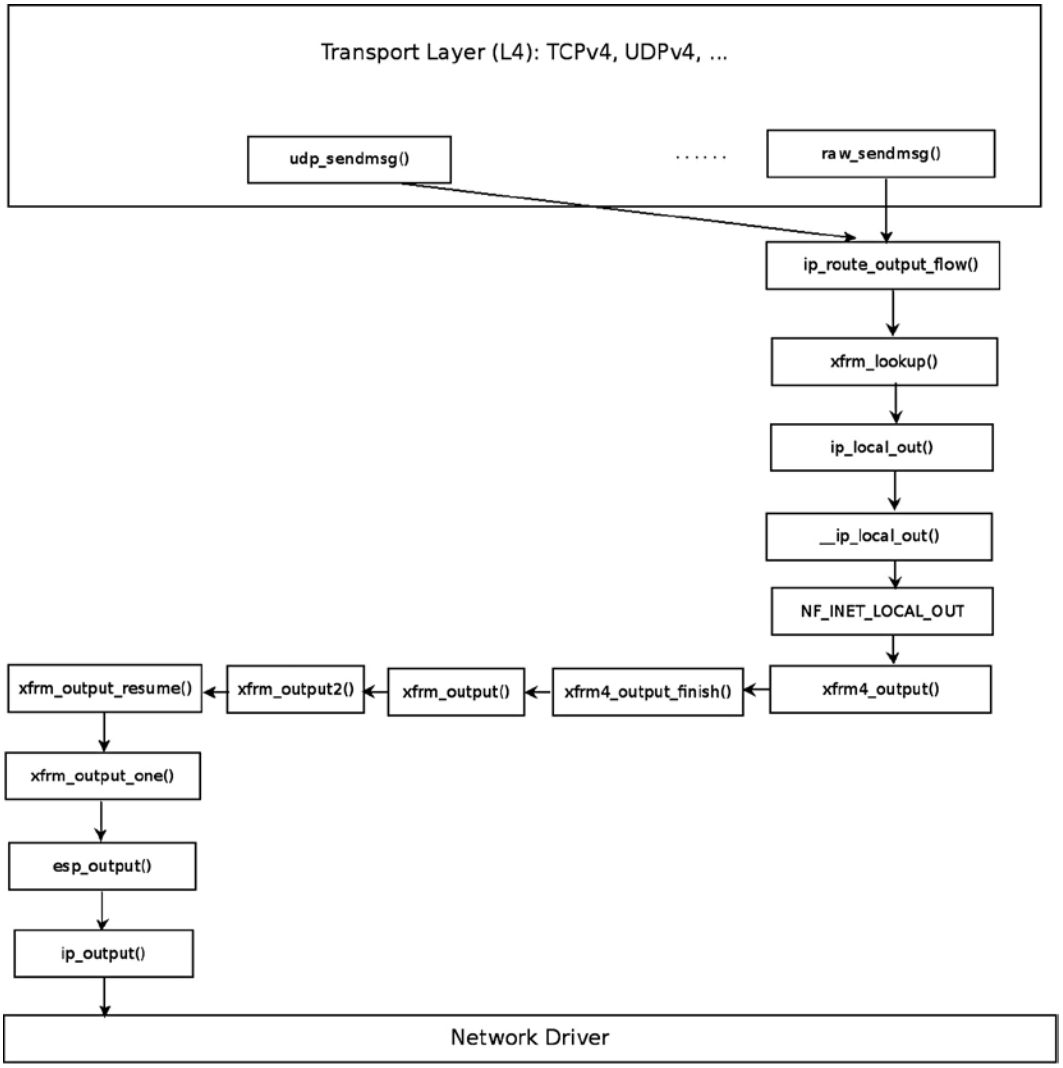

Figure 10-3. *Transmitting IPv4 ESP packet, transport mode. For the sake of simplicity, the case of creating a dummy bundle (when there are no XFRM states) and some other details are omitted*

The following section talks about how a lookup is performed in XFRM.

XFRM Lookup

The `xfrm_lookup()` method is called for each packet that is sent out of the system. You want this lookup to be as efficient as possible. To achieve this goal, bundles are used. Bundles let you cache important information such as the route, the policies, the number of policies, and more; these bundles, which are instances of the `xfrm_dst` structure, are stored by using the flow cache. When the first packet of some flow arrives, you create an entry in the generic flow cache and subsequently create a bundle (`xfrm_dst` object). The bundle creation is done after a lookup for this bundle fails, because it is the first packet of this flow. When subsequent packets of this flow arrive, you will get a hit when performing a flow cache lookup:

```
struct xfrm_dst {
        union {
                struct dst_entry        dst;
                struct rtable           rt;
                struct rt6_info         rt6;
        } u;
        struct dst_entry *route;
        struct flow_cache_object flo;
        struct xfrm_policy *pols[XFRM_POLICY_TYPE_MAX];
        int num_pols, num_xfrms;
#ifdef CONFIG_XFRM_SUB_POLICY
        struct flowi *origin;
        struct xfrm_selector *partner;
#endif
        u32 xfrm_genid;
        u32 policy_genid;
        u32 route_mtu_cached;
        u32 child_mtu_cached;
        u32 route_cookie;
        u32 path_cookie;
};
```

(include/net/xfrm.h)

The `xfrm_lookup()` method is a very complex method. I discuss its important parts but I don't delve into all its nuances. Figure *10-4* shows a block diagram of the internals of the `xfrm_lookup()` method.

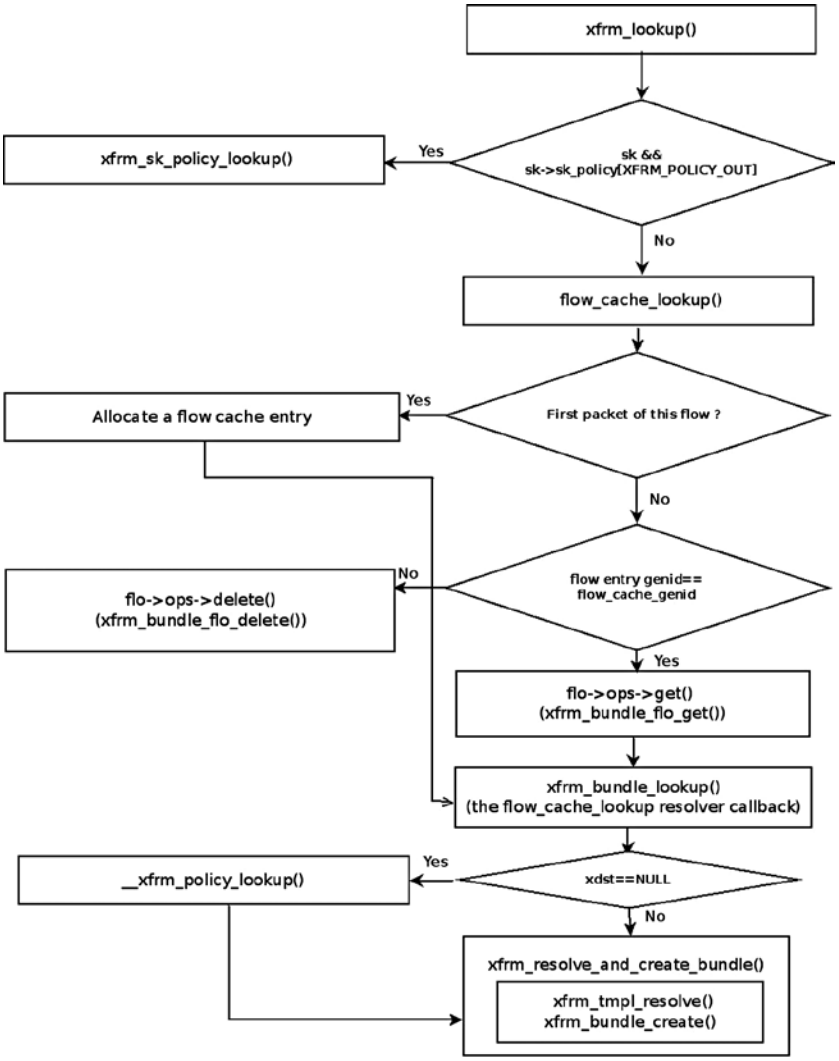

Figure 10-4. *xfrm_lookup( ) internals*

Let's take a look at the `xfrm_lookup()` method:

```
struct dst_entry *xfrm_lookup(struct net *net, struct dst_entry *dst_orig,
                              const struct flowi *fl, struct sock *sk, int flags)
{
```

The `xfrm_lookup()` method handles only the Tx path; so you set the flow direction (`dir`) to be FLOW_DIR_OUT by:

```
        u8 dir = policy_to_flow_dir(XFRM_POLICY_OUT);
```

If a policy is associated with this socket, you perform a lookup by the `xfrm_sk_policy_lookup()` method, which checks whether the packet flow matches the policy selector. Note that if the packet is to be forwarded, the `xfrm_lookup()` method was invoked from the `__xfrm_route_forward()` method, and there is no socket associated with the packet, because it was not generated on the local host; in this case, the specified `sk` argument is NULL:

```
        if (sk && sk->sk_policy[XFRM_POLICY_OUT]) {
                num_pols = 1;
                pols[0] = xfrm_sk_policy_lookup(sk, XFRM_POLICY_OUT, fl);


        . . .
}
```

If there is no policy associated with this socket, you perform a lookup in the generic flow cache by calling the `flow_cache_lookup()` method, passing as an argument a function pointer to the `xfrm_bundle_lookup` method (the `resolver` callback). The key to the lookup is the flow object (the specified `fl` parameter). If you don't find an entry in the flow cache, allocate a new flow cache entry. If you find an entry with the same `genid`, call the `xfrm_bundle_flo_get()` method by invoking `flo->ops->get(flo)`. Eventually, you call the `xfrm_bundle_lookup()` method by invoking the `resolver` callback, which gets the flow object as a parameter (`oldflo`). See the `flow_cache_lookup()` method implementation in `net/core/flow.c`:

```
                flo = flow_cache_lookup(net, fl, family, dir, xfrm_bundle_lookup, dst_orig);
```

Fetch the bundle (`xfrm_dst` object) that contains the flow cache object as a member:

```
                xdst = container_of(flo, struct xfrm_dst, flo);
```

Fetch cached data, like the number of policies, number of templates, the policies and the route:

```
           num_pols = xdst->num_pols;
           num_xfrms = xdst->num_xfrms;
           memcpy(pols, xdst->pols, sizeof(struct xfrm_policy*) * num_pols);
           route = xdst->route;
}

dst = &xdst->u.dst;
```

Next comes handling a dummy bundle. A *dummy bundle* is a bundle where the route member is NULL. It is created in the XFRM bundle lookup process (by the `xfrm_bundle_lookup()` method) when no XFRM states were found, by calling the `xfrm_create_dummy_bundle()` method. In such a case, either one of the two options are available, according to the value of `sysctl_larval_drop` (`/proc/sys/net/core/xfrm_larval_drop`):

- If `sysctl_larval_drop` is set (which means its value is 1—it is so by default, as mentioned earlier in this chapter), the packet should be discarded.
- If `sysctl_larval_drop` is not set (its value is 0), the packets are kept in a per-policy queue (`polq.hold_queue`), which can contain up to 100 (XFRM_MAX_QUEUE_LEN) SKBs; this is implemented by the `xdst_queue_output()` method. These packets are kept until the XFRM

states are resolved or until some timeout elapses. Once the states are resolved, the packets are sent out of the queue. If the XFRM states are not resolved after some time interval (the timeout of the `xfrm_policy_queue` object), the queue is flushed by the `xfrm_queue_purge()` method:

```
if (route == NULL && num_xfrms > 0) {
        /* The only case when xfrm_bundle_lookup() returns a
         * bundle with null route, is when the template could
         * not be resolved. It means policies are there, but
         * bundle could not be created, since we don't yet
         * have the xfrm_state's. We need to wait for KM to
         * negotiate new SA's or bail out with error.*/
         if (net->xfrm.sysctl_larval_drop) {
```

For IPv4, the `make_blackhole()` method calls the `ipv4_blackhole_route()` method. For IPv6, it calls the `ip6_blackhole_route()` method:

```
        return make_blackhole(net, family, dst_orig);
}
```

The next section covers one of the most important features of IPsec—NAT traversal—and explains what it is and why it is needed.

NAT Traversal in IPsec

Why don't NAT devices allow IPsec traffic to pass? NAT changes the IP addresses and sometimes also the port numbers of the packet. As a result, it recalculates the checksum of the TCP or the UDP header. The transport layer checksum calculation takes into account the source and destination of the IP addresses. So even if only the IP addresses were changed, the TCP or UDP checksum should be recalculated. However, with ESP encryption in transport mode, the NAT device can't update the checksum because the TCP or UDP headers are encrypted with ESP. There are protocols where the checksum does not cover the IP header (like SCTP), so this problem does not occur there. To solve these problems, the NAT traversal standard for IPsec was developed (or, as officially termed in RFC 3948, "UDP Encapsulation of IPsec ESP Packets"). UDP Encapsulation can be applied to IPv4 packets as well as to IPv6 packets. NAT traversal solutions are not limited to IPsec traffic; these techniques are typically required for client-to-client networking applications, especially for peer-to-peer and Voice over Internet Protocol (VoIP) applications.

There are some partial solutions for VoIP NAT-traversal, such as STUN, TURN, ICE, and more. I should mention here that strongSwan implements the IKEv2 Mediation Extension service (http://tools.ietf.org/html/draft-brunner-ikev2-mediation-00), which allows two VPN endpoints located behind a NAT router each to establish a direct peer-to-peer IPsec tunnel using a mechanism similar to TURN and ICE. STUN, for example, is used in the VoIP open source Ekiga client (formerly gnomemeeting). The problem with these solutions is NAT devices they don't cope with. Devices called SBCs (session border controllers) provide a full solution for NAT traversal in VoIP. SBCs can be implemented in hardware (Juniper Networks, for example, provides a router-integrated SBC solution) or in software. These SBC solutions perform NAT traversal of the media traffic—which is sent by Real Time Protocol (RTP)—and sometimes also for the signaling traffic—which is sent by Session Initiation Protocol (SIP). NAT traversal is optional in IKEv2. Openswan, strongSwan, and racoon support NAT traversal, but Openswan and racoon support NAT-T only with IKEv1, whereas strongSwan supports NAT traversal in both IKEv1 and IKEv2.

NAT-T Mode of Operation

How does NAT traversal work? First, keep in mind that NAT-T is a good solution only for ESP traffic and not for AH. Another restriction is that NAT-T can't be used with manual keying, but only with IKEv1 and IKEv2. This is because NAT-T is tied with exchanging IKEv1/IKEv2 messages. First, you must tell the userspace daemon (`pluto`) that you want to use the NAT traversal feature, because it is not activated by default. You do that in Openswan by adding `nat_traversal=yes` to the connection parameters in `/etc/ipsec.conf`. Clients not behind a NAT are not affected by the addition of this entry. In strongSwan, the IKEv2 `charon` daemon always supports NAT traversal, and this feature cannot be deactivated. In the first phase of IKE (Main Mode), you check whether both peers support NAT-T. In IKEv1, when a peer supports NAT-T, one of the ISAKAMP header members (vendor ID) tells whether it supports NAT-T. In IKEv2, NAT-T is part of the standard and does not have to be announced. If this condition is met, you check whether there is one or more NAT devices in the path between the two IPsec peers by sending NAT-D payload messages. If this condition is also met, NAT-T protects the original IPsec encoded packet by inserting in it a UDP header between the IP header and the ESP header. Both the source and destination ports in the UDP header are 4500. Besides, NAT-T sends keep-alive messages every 20 seconds so that the NAT retains its mapping. Keep alive messages are also sent on UDP port 4500 and are recognized by their content and value (which is one byte, 0xFF). When this packet reaches the IPsec peer, after going through the NAT, the kernel strips the UDP header and decrypts the ESP payload. See the `xfrm4_udp_encap_rcv()` method in `net/ipv4/xfrm4_input.c`.

Summary

This chapter covered IPsec and the XFRM framework, which is the infrastructure of IPsec, and XFRM policies and states, which are the fundamental data structures of the XFRM framework. I also discussed IKE, the ESP4 implementation, the Rx/Tx path of ESP4 in transport mode, and NAT traversal in IPsec. Chapter 11 deals with the following transport Layer (L4) protocols: UDP, TCP, SCTP, and DCCP. The "Quick Reference" section that follows covers the top methods related to the topics discussed in this chapter, ordered by their context.

Quick Reference

I conclude this chapter with a short list of important methods of IPsec. Some of them were mentioned in this chapter. Afterward, I include a table of XFRM SNMP MIB counters.

Methods

Let's start with the methods.

bool xfrm_selector_match(const struct xfrm_selector *sel, const struct flowi *fl, unsigned short family);

This method returns `true` when the specified flow matches the specified XFRM selector. Invokes the `__xfrm4_selector_match()` method for IPv4 or the `__xfrm6_selector_match()` method for IPv6.

int xfrm_policy_match(const struct xfrm_policy *pol, const struct flowi *fl, u8 type, u16 family, int dir);

This method returns 0 if the specified policy can be applied to the specified flow, otherwise it returns an `-errno`.

struct xfrm_policy *xfrm_policy_alloc(struct net *net, gfp_t gfp);

This method allocates and initializes an XFRM policy. It sets its reference counter to 1, initializes the read-write lock, assigns the policy namespace (`xp_net`) to be the specified network namespace, sets its timer callback to be `xfrm_policy_timer()`, and sets its state resolution packet queue timer (`policy->polq.hold_timer`) callback to be `xfrm_policy_queue_process()`.

void xfrm_policy_destroy(struct xfrm_policy *policy);

This method removes the timer of specified XFRM policy object and releases the specified XFRM policy memory.

void xfrm_pol_hold(struct xfrm_policy *policy);

This method increments by 1 the reference count of the specified XFRM policy.

static inline void xfrm_pol_put(struct xfrm_policy *policy);

This method decrements by 1 the reference count of the specified XFRM policy. If the reference count reaches 0, call the `xfrm_policy_destroy()` method.

struct xfrm_state_afinfo *xfrm_state_get_afinfo(unsigned int family);

This method returns the `xfrm_state_afinfo` object associated with the specified protocol family.

struct dst_entry *xfrm_bundle_create(struct xfrm_policy *policy, struct xfrm_state **xfrm, int nx, const struct flowi *fl, struct dst_entry *dst);

This method creates an XFRM bundle. Called from the `xfrm_resolve_and_create_bundle()` method.

int policy_to_flow_dir(int dir);

This method returns the flow direction according to the specified policy direction. For example, return FLOW_DIR_IN when the specified direction is XFRM_POLICY_IN, and so on.

static struct xfrm_dst *xfrm_create_dummy_bundle(struct net *net, struct dst_entry *dst, const struct flowi *fl, int num_xfrms, u16 family);

This method creates a dummy bundle. Called from the `xfrm_bundle_lookup()` method when policies were found but there are no matching states.

struct xfrm_dst *xfrm_alloc_dst(struct net *net, int family);

This method allocates an XFRM bundle object. Called from the `xfrm_bundle_create()` method and from the `xfrm_create_dummy_bundle()` method.

int xfrm_policy_insert(int dir, struct xfrm_policy *policy, int excl);

This method adds an XFRM policy to the SPD. Invoked from the `xfrm_add_policy()` method (`net/xfrm/xfrm_user.c`), or from the `pfkey_spdadd()` method (`net/key/af_key.c`).

int xfrm_policy_delete(struct xfrm_policy *pol, int dir);

This method releases the resources of the specified XFRM policy object. The direction argument (`dir`) is needed to decrement by 1 the corresponding XFRM policy counter in the `policy_count` in the per namespace `netns_xfrm` object.

int xfrm_state_add(struct xfrm_state *x);

This method adds the specified XFRM state to the SAD.

int xfrm_state_delete(struct xfrm_state *x);

This method deletes the specified XFRM state from the SAD.

void __xfrm_state_destroy(struct xfrm_state *x);

This method releases the resources of an XFRM state by adding it to the XFRM states garbage list and activating the XFRM state garbage collector.

int xfrm_state_walk(struct net *net, struct xfrm_state_walk *walk, int (*func)(struct xfrm_state *, int, void*), void *data);

This method iterates over all XFRM states (`net->xfrm.state_all`) and invokes the specified `func` callback.

struct xfrm_state *xfrm_state_alloc(struct net *net);

This method allocates and initializes an XFRM state.

void xfrm_queue_purge(struct sk_buff_head *list);

This method flushes the state resolution per-policy queue (`polq.hold_queue`).

int xfrm_input(struct sk_buff *skb, int nexthdr, __be32 spi, int encap_type);

This method is the main Rx IPsec handler.

static struct dst_entry *make_blackhole(struct net *net, u16 family, struct dst_entry *dst_orig);

This method is invoked from the `xfrm_lookup()` method when there are no resolved states and `sysctl_larval_drop` is set. For IPv4, the `make_blackhole()` method calls the `ipv4_blackhole_route()` method; for IPv6, it calls the `ip6_blackhole_route()` method.

int xdst_queue_output(struct sk_buff *skb);

This method handles adding packets to the per-policy state resolution packet queue (`pq->hold_queue`). This queue can contain up to 100 (XFRM_MAX_QUEUE_LEN) packets.

struct net *xs_net(struct xfrm_state *x);

This method returns the namespace object (`xs_net`) associated with the specified `xfrm_state` object.

struct net *xp_net(const struct xfrm_policy *xp);

This method returns the namespace object (`xp_net`) associated with the specified `xfrm_policy` object.

int xfrm_policy_id2dir(u32 index);

This method returns the direction of the policy according to the specified index.

int esp_input(struct xfrm_state *x, struct sk_buff *skb);

This method is the main IPv4 ESP protocol handler.

struct ip_esp_hdr *ip_esp_hdr(const struct sk_buff *skb);

This method returns the ESP header associated with the specified SKB.

int verify_newpolicy_info(struct xfrm_userpolicy_info *p);

This method verifies that the specified `xfrm_userpolicy_info` object contains valid values. (`xfrm_userpolicy_info` is the object which is passed from userspace). It returns 0 if it is a valid object, and -EINVAL or -EAFNOSUPPORT if not.

Table

Table 10-1 lists XFRM SNMP MIB counters.

Table 10-1. *XFRM SNMP MIB counters*

Linux Symbol	SNMP (procfs) Symbol	Methods in Which the Counter Might Be Incremented
LINUX_MIB_XFRMINERROR	XfrmInError	xfrm_input()
LINUX_MIB_XFRMINBUFFERERROR	XfrmInBufferError	xfrm_input(),__xfrm_policy_check()
LINUX_MIB_XFRMINHDRERROR	XfrmInHdrError	xfrm_input(),__xfrm_policy_check()
LINUX_MIB_XFRMINNOSTATES	XfrmInNoStates	xfrm_input()
LINUX_MIB_XFRMINSTATEPROTOERROR	XfrmInStateProtoError	xfrm_input()
LINUX_MIB_XFRMINSTATEMODEERROR	XfrmInStateModeError	xfrm_input()
LINUX_MIB_XFRMINSTATESEQERROR	XfrmInStateSeqError	xfrm_input()
LINUX_MIB_XFRMINSTATEEXPIRED	XfrmInStateExpired	xfrm_input()
LINUX_MIB_XFRMINSTATEMISMATCH	XfrmInStateMismatch	xfrm_input(), __xfrm_policy_check()
LINUX_MIB_XFRMINSTATEINVALID	XfrmInStateInvalid	xfrm_input()
LINUX_MIB_XFRMINTMPLMISMATCH	XfrmInTmplMismatch	__xfrm_policy_check()
LINUX_MIB_XFRMINNOPOLS	XfrmInNoPols	__xfrm_policy_check()
LINUX_MIB_XFRMINPOLBLOCK	XfrmInPolBlock	__xfrm_policy_check()
LINUX_MIB_XFRMINPOLERROR	XfrmInPolError	__xfrm_policy_check()
LINUX_MIB_XFRMOUTERROR	XfrmOutError	xfrm_output_one(),xfrm_output()
LINUX_MIB_ XFRMOUTBUNDLEGENERROR	XfrmOutBundleGenError	xfrm_resolve_and_create_bundle()
LINUX_MIB_ XFRMOUTBUNDLECHECKERROR	XfrmOutBundleCheckError	xfrm_resolve_and_create_bundle()
LINUX_MIB_XFRMOUTNOSTATES	XfrmOutNoStates	xfrm_lookup()
LINUX_MIB_ XFRMOUTSTATEPROTOERROR	XfrmOutStateProtoError	xfrm_output_one()
LINUX_MIB_ XFRMOUTSTATEMODEERROR	XfrmOutStateModeError	xfrm_output_one()
LINUX_MIB_XFRMOUTSTATESEQERROR	XfrmOutStateSeqError	xfrm_output_one()
LINUX_MIB_XFRMOUTSTATEEXPIRED	XfrmOutStateExpired	xfrm_output_one()
LINUX_MIB_XFRMOUTPOLBLOCK	XfrmOutPolBlock	xfrm_lookup()
LINUX_MIB_XFRMOUTPOLDEAD	XfrmOutPolDead	n/a
LINUX_MIB_XFRMOUTPOLERROR	XfrmOutPolError	xfrm_bundle_lookup(), xfrm_resolve_and_create_bundle()
LINUX_MIB_XFRMFWDHDRERROR	XfrmFwdHdrError	__xfrm_route_forward()
LINUX_MIB_XFRMOUTSTATEINVALID	XfrmOutStateInvalid	xfrm_output_one()

■ **Note** The IPsec git tree: `git://git.kernel.org/pub/scm/linux/kernel/git/klassert/ipsec.git`.

The ipsec git tree is for fixes for the IPsec networking subsystem; the development in this tree is done against David Miller's net git tree.

The ipsec-next git tree: `git://git.kernel.org/pub/scm/linux/kernel/git/klassert/ipsec-next.git`.

The ipsec-next tree is for changes for IPsec with linux-next as target; the development in this tree is done against David Miller's net-next git tree.

The IPsec subsystem maintainers are Steffen Klassert, Herbert Xu, and David S. Miller.