                   PF_KEY Key Management API, Version 2

Status of this Memo

   This memo provides information for the Internet community. It does
   not specify an Internet standard of any kind. Distribution of this
   memo is unlimited.

Copyright Notice

Abstract

   A generic key management API that can be used not only for IP
   Security [Atk95a] [Atk95b] [Atk95c] but also for other network
   security services is presented in this document. Version 1 of this
   API was implemented inside 4.4-Lite BSD as part of the U. S. Naval
   Research Laboratory's freely distributable and usable IPv6 and IPsec
   implementation[AMPMC96]. It is documented here for the benefit of
   others who might also adopt and use the API, thus providing increased
   portability of key management applications (e.g. a manual keying
   application, an ISAKMP daemon, a GKMP daemon [HM97a][HM97b], a
   Photuris daemon, or a SKIP certificate discovery protocol daemon).

Table of Contents

1 Introduction

   PF_KEY is a new socket protocol family used by trusted privileged key
   management applications to communicate with an operating system's key
   management internals (referred to here as the "Key Engine" or the
   Security Association Database (SADB)). The Key Engine and its
   structures incorporate the required security attributes for a session
   and are instances of the "Security Association" (SA) concept
   described in [Atk95a]. The names PF_KEY and Key Engine thus refer to
   more than cryptographic keys and are retained for consistency with
   the traditional phrase, "Key Management".

   PF_KEY is derived in part from the BSD routing socket, PF_ROUTE.
   [Skl91] This document describes Version 2 of PF_KEY. Version 1 was
   implemented in the first five alpha test versions of the NRL
   IPv6+IPsec Software Distribution for 4.4-Lite BSD UNIX and the Cisco
   ISAKMP/Oakley key management daemon. Version 2 extends and refines
   this interface. Theoretically, the messages defined in this document
   could be used in a non-socket context (e.g. between two directly
   communicating user-level processes), but this document will not
   discuss in detail such possibilities.

   Security policy is deliberately omitted from this interface. PF_KEY
   is not a mechanism for tuning systemwide security policy, nor is it
   intended to enforce any sort of key management policy. The developers
   of PF_KEY believe that it is important to separate security
   mechanisms (such as PF_KEY) from security policies. This permits a
   single mechanism to more easily support multiple policies.

1.1 Terminology

   Even though this document is not intended to be an actual Internet
   standard, the words that are used to define the significance of
   particular features of this interface are usually capitalized. Some
   of these words, including MUST, MAY, and SHOULD, are detailed in
   [Bra97].

   - CONFORMANCE and COMPLIANCE

   Conformance to this specification has the same meaning as compliance
   to this specification. In either case, the mandatory-to-implement,
   or MUST, items MUST be fully implemented as specified here. If any
   mandatory item is not implemented as specified here, that
   implementation is not conforming and not compliant with this
   specification.

This specification also uses many terms that are commonly used in the
context of network security. Other documents provide more
definitions and background information on these [VK83, HA94, Atk95a].
Two terms deserve special mention:

- (Encryption/Authentication) Algorithm

For PF_KEY purposes, an algorithm, whether encryption or
authentication, is the set of operations performed on a packet to
complete authentication or encryption as indicated by the SA type. A
PF_KEY algorithm MAY consist of more than one cryptographic
algorithm. Another possibility is that the same basic cryptographic
algorithm may be applied with different modes of operation or some
other implementation difference. These differences, henceforth called
algorithm differentiators, distinguish between different PF_KEY
algorithms, and options to the same algorithm. Algorithm
differentiators will often cause fundamentally different security
properties.

For example, both DES and 3DES use the same cryptographic algorithm,
but they are used differently and have different security properties.
The triple-application of DES is considered an algorithm
differentiator. There are therefore separate PF_KEY algorithms for
DES and 3DES. Keyed-MD5 and HMAC-MD5 use the same hash function, but
construct their message authentication codes differently. The use of
HMAC is an algorithm differentiator. DES-ECB and DES-CBC are the
same cryptographic algorithm, but use a different mode. Mode (e.g.,
chaining vs. code-book) is an algorithm differentiator. Blowfish with
a 128-bit key, however, is similar to Blowfish with a 384-bit key,
because the algorithm's workings are otherwise the same and therefore
the key length is not an algorithm differentiator.

In terms of IP Security, a general rule of thumb is that whatever
might be labeled the "encryption" part of an ESP transform is
probably a PF_KEY encryption algorithm. Whatever might be labelled
the "authentication" part of an AH or ESP transform is probably a
PF_KEY authentication algorithm.

1.2 Conceptual Model

This section describes the conceptual model of an operating system
that implements the PF_KEY key management application programming
interface. This section is intended to provide background material
useful to understand the rest of this document. Presentation of this
conceptual model does not constrain a PF_KEY implementation to
strictly adhere to the conceptual components discussed in this
subsection.

Key management is most commonly implemented in whole or in part at
the application layer. For example, the ISAKMP/Oakley, GKMP, and
Photuris proposals for IPsec key management are all application-layer
protocols. Manual keying is also done at the application layer.
Even parts of the SKIP IP-layer keying proposal can be implemented at
the application layer. Figure 1 shows the relationship between a Key
Management daemon and PF_KEY. Key management daemons use PF_KEY to
communicate with the Key Engine and use PF_INET (or PF_INET6 in the
case of IPv6) to communicate, via the network, with a remote key
management entity.

The "Key Engine" or "Security Association Database (SADB)" is a
logical entity in the kernel that stores, updates, and deletes
Security Association data for various security protocols. There are
logical interfaces within the kernel (e.g. getassocbyspi(),
getassocbysocket()) that security protocols inside the kernel (e.g.
IP Security, aka IPsec) use to request and obtain Security
Associations.

In the case of IPsec, if by policy a particular outbound packet needs
processing, then the IPsec implementation requests an appropriate
Security Association from the Key Engine via the kernel-internal
interface. If the Key Engine has an appropriate SA, it allocates the
SA to this session (marking it as used) and returns the SA to the
IPsec implementation for use. If the Key Engine has no such SA but a
key management application has previously indicated (via a PF_KEY
SADB_REGISTER message) that it can obtain such SAs, then the Key
Engine requests that such an SA be created (via a PF_KEY SADB_ACQUIRE
message). When the key management daemon creates a new SA, it places
it into the Key Engine for future use.

```
                    +---------------+
                    |Key Mgmt Daemon|
                    +---------------+
                       |         |
                       |         |
                       |         |                    Applications
                 ======[PF_KEY]====[PF_INET]==========================
                       |         |                    OS Kernel
                 +-----------+  +----------------+
                 | Key Engine |  | TCP/IP,        |
                 |  or  SADB  |--| including IPsec |
                 +-----------+  |                |
                                +----------------+
                                   |
                                +-----------+
                                | Network   |
                                | Interface |
                                +-----------+
```
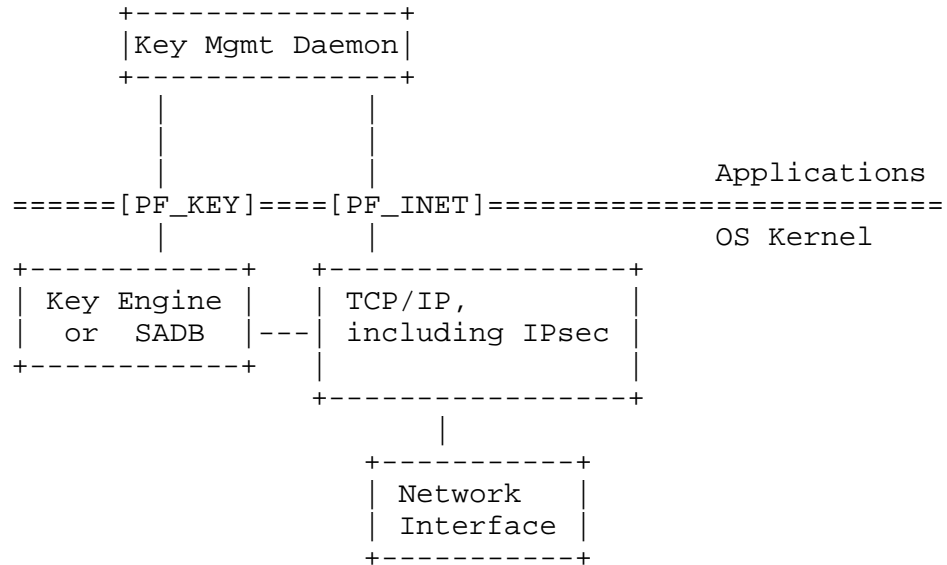
              Figure 1: Relationship of Key Mgmt to PF_KEY

   For performance reasons, some security protocols (e.g. IP Security)
   are usually implemented inside the operating system kernel. Other
   security protocols (e.g. OSPFv2 Cryptographic Authentication) are
   implemented in trusted privileged applications outside the kernel.
   Figure 2 shows a trusted, privileged routing daemon using PF_INET to
   communicate routing information with a remote routing daemon and
   using PF_KEY to request, obtain, and delete Security Associations
   used with a routing protocol.

```
                  +---------------+
                  |Routing  Daemon|
                  +---------------+
                     |          |
                     |          |
                     |          |                     Applications
              ======[PF_KEY]====[PF_INET]=========================
                     |          |                     OS Kernel
              +------------+  +---------+
              | Key Engine |  | TCP/IP  |
              |   or  SADB |--|         |
              +------------+  +---------+
                                  |
                            +-----------+
                            | Network   |
                            | Interface |
                            +-----------+
```
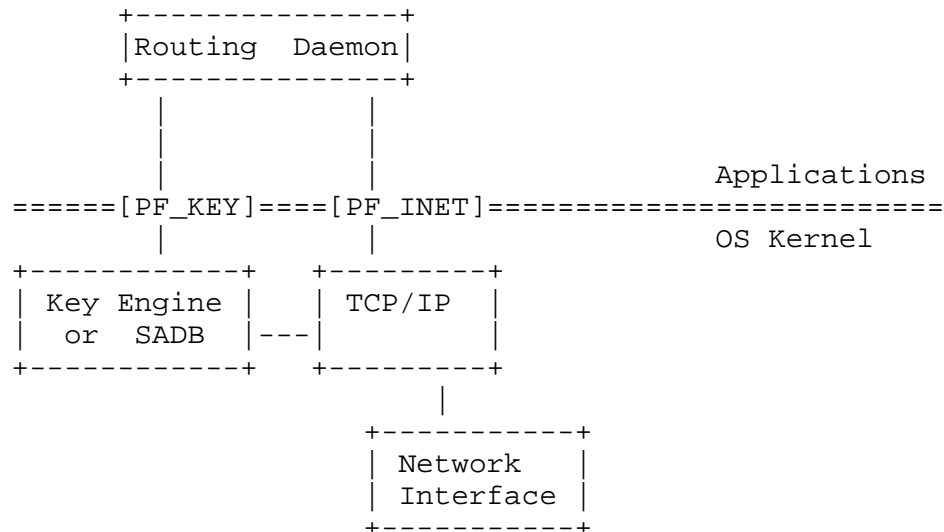
        Figure 2: Relationship of Trusted Application to PF_KEY

   When a trusted privileged application is using the Key Engine but
   implements the security protocol within itself, then operation varies
   slightly. In this case, the application needing an SA sends a PF_KEY
   SADB_ACQUIRE message down to the Key Engine, which then either
   returns an error or sends a similar SADB_ACQUIRE message up to one or
   more key management applications capable of creating such SAs. As
   before, the key management daemon stores the SA into the Key Engine.
   Then, the trusted privileged application uses an SADB_GET message to
   obtain the SA from the Key Engine.

   In some implementations, policy may be implemented in user-space,
   even though the actual cryptographic processing takes place in the
   kernel. Such policy communication between the kernel mechanisms and
   the user-space policy MAY be implemented by PF_KEY extensions, or
   other such mechanism. This document does not specify such
   extensions. A PF_KEY implementation specified by the memo does NOT
   have to support configuring systemwide policy using PF_KEY.

   Untrusted clients, for example a user's web browser or telnet client,
   do not need to use PF_KEY. Mechanisms not specified here are used by
   such untrusted client applications to request security services (e.g.
   IPsec) from an operating system. For security reasons, only trusted,
   privileged applications are permitted to open a PF_KEY socket.

1.3 PF_KEY Socket Definition

   The PF_KEY protocol family (PF_KEY) symbol is defined in
   <sys/socket.h> in the same manner that other protocol families are
   defined. PF_KEY does not use any socket addresses. Applications
   using PF_KEY MUST NOT depend on the availability of a symbol named
   AF_KEY, but kernel implementations are encouraged to define that
   symbol for completeness.

      The key management socket is created as follows:

      #include <sys/types.h>
      #include <sys/socket.h>
      #include <net/pfkeyv2.h>

      int s;
      s = socket(PF_KEY, SOCK_RAW, PF_KEY_V2);

   The PF_KEY domain currently supports only the SOCK_RAW socket type.
   The protocol field MUST be set to PF_KEY_V2, or else EPROTONOSUPPORT
   will be returned. Only a trusted, privileged process can create a
   PF_KEY socket. On conventional UNIX systems, a privileged process is
   a process with an effective userid of zero. On non-MLS proprietary
   operating systems, the notion of a "privileged process" is
   implementation-defined. On Compartmented Mode Workstations (CMWs) or
   other systems that claim to provide Multi-Level Security (MLS), a
   process MUST have the "key management privilege" in order to open a
   PF_KEY socket[DIA]. MLS systems that don't currently have such a
   specific privilege MUST add that special privilege and enforce it
   with PF_KEY in order to comply and conform with this specification.
   Some systems, most notably some popular personal computers, do not
   have the concept of an unprivileged user. These systems SHOULD take
   steps to restrict the programs allowed to access the PF_KEY API.

1.4 Overview of PF_KEY Messaging Behavior

   A process interacts with the key engine by sending and receiving
   messages using the PF_KEY socket. Security association information
   can be inserted into and retrieved from the kernel's security
   association table using a set of predefined messages. In the normal
   case, all properly-formed messages sent to the kernel are returned to
   all open PF_KEY sockets, including the sender. Improperly formed
   messages will result in errors, and an implementation MUST check for
   a properly formed message before returning it to the appropriate
   listeners. Unlike the routing socket, most errors are sent in reply
   messages, not the errno field when write() or send() fails. PF_KEY
   message delivery is not guaranteed, especially in cases where kernel
   or socket buffers are exhausted and messages are dropped.

Some messages are generated by the operating system to indicate that
actions need to be taken, and are not necessarily in response to any
message sent down by the user. Such messages are not received by all
PF_KEY sockets, but by sockets which have indicated that kernel-
originated messages are to be received. These messages are special
because of the expected frequency at which they will occur. Also, an
implementation may further wish to restrict return messages from the
kernel, in cases where not all PF_KEY sockets are in the same trust
domain.

Many of the normal BSD socket calls have undefined behavior on PF_KEY
sockets. These include: bind(), connect(), socketpair(), accept(),
getpeername(), getsockname(), ioctl(), and listen().

1.5 Common PF_KEY Operations

There are two basic ways to add a new Security Association into the
kernel. The simplest is to send a single SADB_ADD message,
containing all of the SA information, from the application into the
kernel's Key Engine. This approach works particularly well with
manual key management, which is required for IPsec, and other
security protocols.

The second approach to add a new Security Association into the kernel
is for the application to first request a Security Parameters Index
(SPI) value from the kernel using the SADB_GETSPI message and then
send an SADB_UPDATE message with the complete Security Association
data. This second approach works well with key management daemons
when the SPI values need to be known before the entire Security
Association data is known (e.g. so the SPI value can be indicated to
the remote end of the key management session).

An individual Security Association can be deleted using the
SADB_DELETE message. Categories of SAs or the entire kernel SA table
can be deleted using the SADB_FLUSH message.

The SADB_GET message is used by a trusted application-layer process
(e.g. routed(8) or gated(8)) to retrieve an SA (e.g. RIP SA or OSPF
SA) from the kernel's Key Engine.

The kernel or an application-layer can use the SADB_ACQUIRE message
to request that a Security Association be created by some
application-layer key management process that has registered with the
kernel via an SADB_REGISTER message. This ACQUIRE message will have
a sequence number associated with it. This sequence number MUST be
used by followup SADB_GETSPI, SADB_UPDATE, and SADB_ADD messages, in
order to keep track of which request gets its keying material. The
sequence number (described below) is similar to a transaction ID in a

remote procedure call.

The SADB_EXPIRE message is sent from the kernel to key management
applications when the "soft lifetime" or "hard lifetime" of a
Security Association has expired. Key management applications should
use receipt of a soft lifetime SADB_EXPIRE message as a hint to
negotiate a replacement SA so the replacement SA will be ready and in
the kernel before it is needed.

A SADB_DUMP message is also defined, but this is primarily intended
for PF_KEY implementor debugging and is not used in ordinary
operation of PF_KEY.

1.6 Differences Between PF_KEY and PF_ROUTE

The following bullets are points of difference between the routing
socket and PF_KEY. Programmers who are used to the routing socket
semantics will find some differences in PF_KEY.

* PF_KEY message errors are usually returned in PF_KEY messages
  instead of causing write() operations to fail and returning the
  error number in errno. This means that other listeners on a PF_KEY
  socket can be aware that requests from another process failed,
  which can be useful for auditing purposes. This also means that
  applications that fail to read PF_KEY messages cannot do error
  checking.

  An implementation MAY return the errors EINVAL, ENOMEM, and ENOBUFS
  by causing write() operations to fail and returning the error
  number in errno. This is an optimization for common error cases in
  which it does not make sense for any other process to receive the
  error. An application MUST NOT depend on such errors being set by
  the write() call, but it SHOULD check for such errors, and handle
  them in an appropriate manner.

 * The entire message isn't always reflected in the reply. A SADB_ADD
   message is an example of this.

 * The PID is not set by the kernel. The process that originates the
   message MUST set the sadb_msg_pid to its own PID. If the kernel
   ORIGINATES a message, it MUST set the sadb_msg_pid to 0. A reply
   to an original message SHOULD have the pid of the original message.
   (E.g. the kernel's response to an SADB_ADD SHOULD have its pid set
   to the pid value of the original SADB_ADD message.)

1.7 Name Space

   All PF_KEYv2 preprocessor symbols and structure definitions are
   defined as a result of including the header file <net/pfkeyv2.h>.
   There is exactly one exception to this rule: the symbol "PF_KEY" (two
   exceptions if "AF_KEY" is also counted), which is defined as a result
   of including the header file <sys/socket.h>. All PF_KEYv2
   preprocessor symbols start with the prefix "SADB_" and all structure
   names start with "sadb_". There are exactly two exceptions to this
   rule: the symbol "PF_KEY_V2" and the symbol "PFKEYV2_REVISION".

   The symbol "PFKEYV2_REVISION" is a date-encoded value not unlike
   certain values defined by POSIX and X/Open. The current value for
   PFKEYV2_REVISION is 199806L, where 1998 is the year and 06 is the
   month.

   Inclusion of the file <net/pfkeyv2.h> MUST NOT define symbols or
   structures in the PF_KEYv2 name space that are not described in this
   document without the explicit prior permission of the authors. Any
   symbols or structures in the PF_KEYv2 name space that are not
   described in this document MUST start with "SADB_X_" or "sadb_x_". An
   implementation that fails to obey these rules IS NOT COMPLIANT WITH
   THIS SPECIFICATION and MUST NOT make any claim to be. These rules
   also apply to any files that might be included as a result of
   including the file <net/pfkeyv2.h>. This rule provides implementors
   with some assurance that they will not encounter namespace-related
   surprises.

1.8 On Manual Keying

   Not unlike the 4.4-Lite BSD PF_ROUTE socket, this interface allows an
   application full-reign over the security associations in a kernel
   that implements PF_KEY. A PF_KEY implementation MUST have some sort
   of manual interface to PF_KEY, which SHOULD allow all of the
   functionality of the programmatic interface described here.

2. PF_KEY Message Format

   PF_KEY messages consist of a base header followed by additional data
   fields, some of which may be optional. The format of the additional
   data is dependent on the type of message.

   PF_KEY messages currently do not mandate any specific ordering for
   non-network multi-octet fields. Unless otherwise specified (e.g. SPI
   values), fields MUST be in host-specific byte order.

2.1 Base Message Header Format

   PF_KEY messages consist of the base message header followed by
   security association specific data whose types and lengths are
   specified by a generic type-length encoding.

   This base header is shown below, using POSIX types. The fields are
   arranged primarily for alignment, and where possible, for reasons of
   clarity.

```
        struct sadb_msg {
                uint8_t sadb_msg_version;
                uint8_t sadb_msg_type;
                uint8_t sadb_msg_errno;
                uint8_t sadb_msg_satype;
                uint16_t sadb_msg_len;
                uint16_t sadb_msg_reserved;
                uint32_t sadb_msg_seq;
                uint32_t sadb_msg_pid;
        };
        /* sizeof(struct sadb_msg) == 16 */
```

   sadb_msg_version
                The version field of this PF_KEY message. This MUST
                be set to PF_KEY_V2. If this is not set to PF_KEY_V2,
                the write() call MAY fail and return EINVAL.
                Otherwise, the behavior is undetermined, given that
                the application might not understand the formatting
                of the messages arriving from the kernel.

   sadb_msg_type   Identifies the type of message. The valid message
                    types are described later in this document.

   sadb_msg_errno   Should be set to zero by the sender. The responder
                    stores the error code in this field if an error has
                    occurred. This includes the case where the responder
                    is in user space. (e.g. user-space negotiation
                    fails, an errno can be returned.)

   sadb_msg_satype Indicates the type of security association(s). Valid
                    Security Association types are declared in the file
                    <net/pfkeyv2.h>. The current set of Security
                    Association types is enumerated later in this
                    document.

sadb_msg_len   Contains the total length, in 64-bit words, of all
                    data in the PF_KEY message including the base header
                    length and additional data after the base header, if
                    any. This length includes any padding or extra space
                    that might exist. Unless otherwise stated, all other
                    length fields are also measured in 64-bit words.

                    On user to kernel messages, this field MUST be
                    verified against the length of the inbound message.
                    EMSGSIZE MUST be returned if the verification fails.
                    On kernel to user messages, a size mismatch is most
                    likely the result of the user not providing a large
                    enough buffer for the message. In these cases, the
                    user application SHOULD drop the message, but it MAY
                    try and extract what information it can out of the
                    message.

    sadb_msg_reserved
                    Reserved value. It MUST be zeroed by the sender. All
                    fields labeled reserved later in the document have
                    the same semantics as this field.

sadb_msg_seq   Contains the sequence number of this message. This
                    field, along with sadb_msg_pid, MUST be used to
                    uniquely identify requests to a process. The sender
                    is responsible for filling in this field. This
                    responsibility also includes matching the
                    sadb_msg_seq of a request (e.g. SADB_ACQUIRE).

                    This field is similar to a transaction ID in a
                    remote procedure call implementation.

sadb_msg_pid   Identifies the process which originated this message,
                    or which process a message is bound for. For
                    example, if process id 2112 sends an SADB_UPDATE
                    message to the kernel, the process MUST set this
                    field to 2112 and the kernel will set this field
                    to 2112 in its reply to that SADB_UPDATE
                    message. This field, along with sadb_msg_seq, can
                    be used to uniquely identify requests to a
                    process.

                    It is currently assumed that a 32-bit quantity will
                    hold an operating system's process ID space.

2.2 Alignment of Headers and Extension Headers

   The base message header is a multiple of 64 bits and fields after it
   in memory will be 64 bit aligned if the base itself is 64 bit
   aligned. Some of the subsequent extension headers have 64 bit fields
   in them, and as a consequence need to be 64 bit aligned in an
   environment where 64 bit quantities need to be 64 bit aligned.

   The basic unit of alignment and length in PF_KEY Version 2 is 64
   bits. Therefore:

      * All extension headers, inclusive of the sadb_ext overlay fields,
        MUST be a multiple of 64 bits long.

      * All variable length data MUST be padded appropriately such that
        its length in a message is a multiple of 64 bits.

      * All length fields are, unless otherwise specified, in units of
        64 bits.

      * Implementations may safely access quantities of between 8 and 64
        bits directly within a message without risk of alignment faults.

   All PF_KEYv2 structures are packed and already have all intended
   padding. Implementations MUST NOT insert any extra fields, including
   hidden padding, into any structure in this document. This forbids
   implementations from "extending" or "enhancing" existing headers
   without changing the extension header type. As a guard against such
   insertion of silent padding, each structure in this document is
   labeled with its size in bytes. The size of these structures in an
   implementation MUST match the size listed.

2.3 Additional Message Fields

   The additional data following the base header consists of various
   length-type-values fields. The first 32-bits are of a constant form:

```
        struct sadb_ext {
                uint16_t sadb_ext_len;
                uint16_t sadb_ext_type;
        };
        /* sizeof(struct sadb_ext) == 4 */
```

   sadb_ext_len   Length of the extension header in 64 bit words,
                   inclusive.

sadb_ext_type   The type of extension header that follows. Values for
                 this field are detailed later. The value zero is
                 reserved.

Types of extension headers include: Association, Lifetime(s),
Address(s), Key(s), Identity(ies), Sensitivity, Proposal, and
Supported. There MUST be only one instance of a extension type in a
message. (e.g. Base, Key, Lifetime, Key is forbidden). An EINVAL
will be returned if there are duplicate extensions within a message.
Implementations MAY enforce ordering of extensions in the order
presented in the EXTENSION HEADER VALUES section.

If an unknown extension type is encountered, it MUST be ignored.
Applications using extension headers not specified in this document
MUST be prepared to work around other system components not
processing those headers. Likewise, if an application encounters an
unknown extension from the kernel, it must be prepared to work around
it. Also, a kernel that generates extra extension header types MUST
NOT _depend_ on applications also understanding extra extension
header types.

All extension definitions include these two fields (len and exttype)
because they are instances of a generic extension (not unlike
sockaddr_in and sockaddr_in6 are instances of a generic sockaddr).
The sadb_ext header MUST NOT ever be present in a message without at
least four bytes of extension header data following it, and,
therefore, there is no problem with it being only four bytes long.

All extensions documented in this section MUST be implemented by a
PF_KEY implementation.

2.3.1 Association Extension

The Association extension specifies data specific to a single
security association. The only times this extension is not present is
when control messages (e.g. SADB_FLUSH or SADB_REGISTER) are being
passed and on the SADB_ACQUIRE message.

```
        struct sadb_sa {
                uint16_t sadb_sa_len;
                uint16_t sadb_sa_exttype;
                uint32_t sadb_sa_spi;
                uint8_t sadb_sa_replay;
                uint8_t sadb_sa_state;
                uint8_t sadb_sa_auth;
                uint8_t sadb_sa_encrypt;
                uint32_t sadb_sa_flags;
        };
```

```
         /* sizeof(struct sadb_sa) == 16 */
```

    sadb_sa_spi   The Security Parameters Index value for the security
                       association. Although this is a 32-bit field, some
                       types of security associations might have an SPI or
                       key identifier that is less than 32-bits long. In
                       this case, the smaller value shall be stored in the
                       least significant bits of this field and the unneeded
                       bits shall be zero. This field MUST be in network
                       byte order.

    sadb_sa_replay   The size of the replay window, if not zero. If zero,
                       then no replay window is in use.

    sadb_sa_state   The state of the security association. The currently
                       defined states are described later in this document.

    sadb_sa_auth   The authentication algorithm to be used with this
                       security association. The valid authentication
                       algorithms are described later in this document. A
                       value of zero means that no authentication is used
                       for this security association.

    sadb_sa_encrypt   The encryption algorithm to be used with this
                       security association. The valid encryption algorithms
                       are described later in this document. A value of zero
                       means that no encryption is used for this security
                       association.

    sadb_sa_flags   A bitmap of options defined for the security
                       association. The currently defined flags are
                       described later in this document.

    The kernel MUST check these values where appropriate. For example,
    IPsec AH with no authentication algorithm is probably an error.

    When used with some messages, the values in some fields in this
    header should be ignored.

2.3.2 Lifetime Extension

    The Lifetime extension specifies one or more lifetime variants for
    this security association. If no Lifetime extension is present the
    association has an infinite lifetime. An association SHOULD have a
    lifetime of some sort associated with it. Lifetime variants come in
    three varieties, HARD - indicating the hard-limit expiration, SOFT -
    indicating the soft-limit expiration, and CURRENT - indicating the
    current state of a given security association. The Lifetime

extension looks like:

```
struct sadb_lifetime {
        uint16_t sadb_lifetime_len;
        uint16_t sadb_lifetime_exttype;
        uint32_t sadb_lifetime_allocations;
        uint64_t sadb_lifetime_bytes;
        uint64_t sadb_lifetime_addtime;
        uint64_t sadb_lifetime_usetime;
};
/* sizeof(struct sadb_lifetime) == 32 */
```

sadb_lifetime_allocations
        For CURRENT, the number of different connections,
        endpoints, or flows that the association has been
        allocated towards. For HARD and SOFT, the number of
        these the association may be allocated towards
        before it expires. The concept of a connection,
        flow, or endpoint is system specific.

sadb_lifetime_bytes
        For CURRENT, how many bytes have been processed
        using this security association. For HARD and SOFT,
        the number of bytes that may be processed using
        this security association before it expires.

sadb_lifetime_addtime
        For CURRENT, the time, in seconds, when the
        association was created. For HARD and SOFT, the
        number of seconds after the creation of the
        association until it expires.

        For such time fields, it is assumed that 64-bits is
        sufficiently large to hold the POSIX time_t value.
        If this assumption is wrong, this field will have to
        be revisited.

sadb_lifetime_usetime
        For CURRENT, the time, in seconds, when association
        was first used. For HARD and SOFT, the number of
        seconds after the first use of the association until
        it expires.

The semantics of lifetimes are inclusive-OR, first-to-expire. This
means that if values for bytes and time, or multiple times, are
passed in, the first of these values to be reached will cause a
lifetime expiration.

2.3.3 Address Extension

   The Address extension specifies one or more addresses that are
   associated with a security association. Address extensions for both
   source and destination MUST be present when an Association extension
   is present. The format of an Address extension is:

           struct sadb_address {
                   uint16_t sadb_address_len;
                   uint16_t sadb_address_exttype;
                   uint8_t sadb_address_proto;
                   uint8_t sadb_address_prefixlen;
                   uint16_t sadb_address_reserved;
           };
           /* sizeof(struct sadb_address) == 8 */

           /* followed by some form of struct sockaddr */

   The sockaddr structure SHOULD conform to the sockaddr structure of
   the system implementing PF_KEY. If the system has an sa_len field, so
   SHOULD the sockaddrs in the message. If the system has NO sa_len
   field, the sockaddrs SHOULD NOT have an sa_len field. All non-address
   information in the sockaddrs, such as sin_zero for AF_INET sockaddrs,
   and sin6_flowinfo for AF_INET6 sockaddrs, MUST be zeroed out. The
   zeroing of ports (e.g. sin_port and sin6_port) MUST be done for all
   messages except for originating SADB_ACQUIRE messages, which SHOULD
   fill them in with ports from the relevant TCP or UDP session which
   generates the ACQUIRE message. If the ports are non-zero, then the
   sadb_address_proto field, normally zero, MUST be filled in with the
   transport protocol's number. If the sadb_address_prefixlen is non-
   zero, then the address has a prefix (often used in KM access control
   decisions), with length specified in sadb_address_prefixlen. These
   additional fields may be useful to KM applications.

   The SRC and DST addresses for a security association MUST be in the
   same protocol family and MUST always be present or absent together in
   a message. The PROXY address MAY be in a different protocol family,
   and for most security protocols, represents an actual originator of a
   packet. (For example, the inner-packets's source address in a
   tunnel.)

   The SRC address MUST be a unicast or unspecified (e.g., INADDR_ANY)
   address. The DST address can be any valid destination address
   (unicast, multicast, or even broadcast). The PROXY address SHOULD be
   a unicast address (there are experimental security protocols where
   PROXY semantics may be different than described above).

2.3.4 Key Extension

   The Key extension specifies one or more keys that are associated with
   a security association. A Key extension will not always be present
   with messages, because of security risks. The format of a Key
   extension is:

            struct sadb_key {
                    uint16_t sadb_key_len;
                    uint16_t sadb_key_exttype;
                    uint16_t sadb_key_bits;
                    uint16_t sadb_key_reserved;
            };
            /* sizeof(struct sadb_key) == 8 */

            /* followed by the key data */

   sadb_key_bits   The length of the valid key data, in bits. A value of
                    zero in sadb_key_bits MUST cause an error.

   The key extension comes in two varieties. The AUTH version is used
   with authentication keys (e.g. IPsec AH, OSPF MD5) and the ENCRYPT
   version is used with encryption keys (e.g. IPsec ESP). PF_KEY deals
   only with fully formed cryptographic keys, not with "raw key
   material". For example, when ISAKMP/Oakley is in use, the key
   management daemon is always responsible for transforming the result
   of the Diffie-Hellman computation into distinct fully formed keys
   PRIOR to sending those keys into the kernel via PF_KEY. This rule is
   made because PF_KEY is designed to support multiple security
   protocols (not just IP Security) and also multiple key management
   schemes including manual keying, which does not have the concept of
   "raw key material". A clean, protocol-independent interface is
   important for portability to different operating systems as well as
   for portability to different security protocols.

   If an algorithm defines its key to include parity bits (e.g. DES)
   then the key used with PF_KEY MUST also include those parity bits.
   For example, this means that a single DES key is always a 64-bit
   quantity.

   When a particular security protocol only requires one authentication
   and/or one encryption key, the fully formed key is transmitted using
   the appropriate key extension. When a particular security protocol
   requires more than one key for the same function (e.g. Triple-DES
   using 2 or 3 keys, and asymmetric algorithms), then those two fully
   formed keys MUST be concatenated together in the order used for
   outbound packet processing. In the case of multiple keys, the
   algorithm MUST be able to determine the lengths of the individual

keys based on the information provided. The total key length (when
combined with knowledge of the algorithm in use) usually provides
sufficient information to make this determination.

Keys are always passed through the PF_KEY interface in the order that
they are used for outbound packet processing. For inbound processing,
the correct order that keys are used might be different from this
canonical concatenation order used with the PF_KEY interface. It is
the responsibility of the implementation to use the keys in the
correct order for both inbound and outbound processing.

For example, consider a pair of nodes communicating unicast using an
ESP three-key Triple-DES Security Association. Both the outbound SA
on the sender node, and the inbound SA on the receiver node will
contain key-A, followed by key-B, followed by key-C in their
respective ENCRYPT key extensions. The outbound SA will use key-A
first, followed by key-B, then key-C when encrypting. The inbound SA
will use key-C, followed by key-B, then key-A when decrypting.
(NOTE: We are aware that 3DES is actually encrypt-decrypt-encrypt.)
The canonical ordering of key-A, key-B, key-C is used for 3DES, and
should be documented. The order of "encryption" is the canonical
order for this example. [Sch96]

The key data bits are arranged most-significant to least significant.
For example, a 22-bit key would take up three octets, with the least
significant two bits not containing key material. Five additional
octets would then be used for padding to the next 64-bit boundary.

While not directly related to PF_KEY, there is a user interface issue
regarding odd-digit hexadecimal representation of keys. Consider the
example of the 16-bit number:

        0x123

That will require two octets of storage. In the absence of other
information, however, unclear whether the value shown is stored as:

        01 23   OR   12 30

It is the opinion of the authors that the former (0x123 == 0x0123) is
the better way to interpret this ambiguity. Extra information (for
example, specifying 0x0123 or 0x1230, or specifying that this is only
a twelve-bit number) would solve this problem.

2.3.5 Identity Extension

   The Identity extension contains endpoint identities. This
   information is used by key management to select the identity
   certificate that is used in negotiations. This information may also
   be provided by a kernel to network security aware applications to
   identify the remote entity, possibly for access control purposes. If
   this extension is not present, key management MUST assume that the
   addresses in the Address extension are the only identities for this
   Security Association. The Identity extension looks like:

           struct sadb_ident {
                   uint16_t sadb_ident_len;
                   uint16_t sadb_ident_exttype;
                   uint16_t sadb_ident_type;
                   uint16_t sadb_ident_reserved;
                   uint64_t sadb_ident_id;
           };
           /* sizeof(struct sadb_ident) == 16 */

           /* followed by the identity string, if present */

   sadb_ident_type The type of identity information that follows.
                   Currently defined identity types are described later
                   in this document.

   sadb_ident_id   An identifier used to aid in the construction of an
                   identity string if none is present. A POSIX user id
                   value is one such identifier that will be used in this
                   field. Use of this field is described later in this
                   document.

   A C string containing a textual representation of the identity
   information optionally follows the sadb_ident extension. The format
   of this string is determined by the value in sadb_ident_type, and is
   described later in this document.

2.3.6 Sensitivity Extension

   The Sensitivity extension contains security labeling information for
   a security association. If this extension is not present, no
   sensitivity-related data can be obtained from this security
   association. If this extension is present, then the need for
   explicit security labeling on the packet is obviated.

           struct sadb_sens {
                   uint16_t sadb_sens_len;
                   uint16_t sadb_sens_exttype;

```
                    uint32_t sadb_sens_dpd;
                    uint8_t sadb_sens_sens_level;
                    uint8_t sadb_sens_sens_len;
                    uint8_t sadb_sens_integ_level;
                    uint8_t sadb_sens_integ_len;
                    uint32_t sadb_sens_reserved;
            };
            /* sizeof(struct sadb_sens) == 16 */

            /* followed by:
                    uint64_t sadb_sens_bitmap[sens_len];
                    uint64_t sadb_integ_bitmap[integ_len]; */
```

```
    sadb_sens_dpd      Describes the protection domain, which allows
                       interpretation of the levels and compartment
                       bitmaps.
    sadb_sens_sens_level
                       The sensitivity level.
    sadb_sens_sens_len
                       The length, in 64 bit words, of the sensitivity
                       bitmap.
    sadb_sens_integ_level
                       The integrity level.
    sadb_sens_integ_len
                       The length, in 64 bit words, of the integrity
                       bitmap.
```

   This sensitivity extension is designed to support the Bell-LaPadula
   [BL74] security model used in compartmented-mode or multi-level
   secure systems, the Clark-Wilson [CW87] commercial security model,
   and/or the Biba integrity model [Biba77]. These formal models can be
   used to implement a wide variety of security policies. The definition
   of a particular security policy is outside the scope of this
   document. Each of the bitmaps MUST be padded to a 64-bit boundary if
   they are not implicitly 64-bit aligned.

2.3.7 Proposal Extension

   The Proposal extension contains a "proposed situation" of algorithm
   preferences. It looks like:

```
            struct sadb_prop {
                    uint16_t sadb_prop_len;
                    uint16_t sadb_prop_exttype;
                    uint8_t sadb_prop_replay;
                    uint8_t sadb_prop_reserved[3];
            };
            /* sizeof(struct sadb_prop) == 8 */
```

```
        /* followed by:
           struct sadb_comb sadb_combs[(sadb_prop_len *
               sizeof(uint64_t) - sizeof(struct sadb_prop)) /
               sizeof(struct sadb_comb)]; */
```

   Following the header is a list of proposed parameter combinations in
   preferential order. The values in these fields have the same
   definition as the fields those values will move into if the
   combination is chosen.

        NOTE: Some algorithms in some security protocols will have
              variable IV lengths per algorithm. Variable length IVs
              are not supported by PF_KEY v2. If they were, however,
              proposed IV lengths would go in the Proposal Extension.

   These combinations look like:

```
        struct sadb_comb {
                uint8_t sadb_comb_auth;
                uint8_t sadb_comb_encrypt;
                uint16_t sadb_comb_flags;
                uint16_t sadb_comb_auth_minbits;
                uint16_t sadb_comb_auth_maxbits;
                uint16_t sadb_comb_encrypt_minbits;
                uint16_t sadb_comb_encrypt_maxbits;
                uint32_t sadb_comb_reserved;
                uint32_t sadb_comb_soft_allocations;
                uint32_t sadb_comb_hard_allocations;
                uint64_t sadb_comb_soft_bytes;
                uint64_t sadb_comb_hard_bytes;
                uint64_t sadb_comb_soft_addtime;
                uint64_t sadb_comb_hard_addtime;
                uint64_t sadb_comb_soft_usetime;
                uint64_t sadb_comb_hard_usetime;
        };

        /* sizeof(struct sadb_comb) == 72 */
```

   sadb_comb_auth If this combination is accepted, this will be the
                   value of sadb_sa_auth.

   sadb_comb_encrypt
                   If this combination is accepted, this will be the
                   value of sadb_sa_encrypt.

    sadb_comb_auth_minbits;
    sadb_comb_auth_maxbits;
                    The minimum and maximum acceptable authentication
                    key lengths, respectably, in bits. If sadb_comb_auth
                    is zero, both of these values MUST be zero. If
                    sadb_comb_auth is nonzero, both of these values MUST
                    be nonzero. If this combination is accepted, a value
                    between these (inclusive) will be stored in the
                    sadb_key_bits field of KEY_AUTH. The minimum MUST
                    NOT be greater than the maximum.

    sadb_comb_encrypt_minbits;
    sadb_comb_encrypt_maxbits;
                    The minimum and maximum acceptable encryption key
                    lengths, respectably, in bits. If sadb_comb_encrypt
                    is zero, both of these values MUST be zero. If
                    sadb_comb_encrypt is nonzero, both of these values
                    MUST be nonzero. If this combination is accepted, a
                    value between these (inclusive) will be stored in
                    the sadb_key_bits field of KEY_ENCRYPT. The minimum
                    MUST NOT be greater than the maximum.

    sadb_comb_soft_allocations
    sadb_comb_hard_allocations
                    If this combination is accepted, these are proposed
                    values of sadb_lifetime_allocations in the SOFT and
                    HARD lifetimes, respectively.

    sadb_comb_soft_bytes
    sadb_comb_hard_bytes
                    If this combination is accepted, these are proposed
                    values of sadb_lifetime_bytes in the SOFT and HARD
                    lifetimes, respectively.

    sadb_comb_soft_addtime
    sadb_comb_hard_addtime
                    If this combination is accepted, these are proposed
                    values of sadb_lifetime_addtime in the SOFT and HARD
                    lifetimes, respectively.

    sadb_comb_soft_usetime
    sadb_comb_hard_usetime
                    If this combination is accepted, these are proposed
                    values of sadb_lifetime_usetime in the SOFT and HARD
                    lifetimes, respectively.

Each combination has an authentication and encryption algorithm,
which may be 0, indicating none. A combination's flags are the same
as the flags in the Association extension. The minimum and maximum
key lengths (which are in bits) are derived from possible a priori
policy decisions, along with basic properties of the algorithm.
Lifetime attributes are also included in a combination, as some
algorithms may know something about their lifetimes and can suggest
lifetime limits.

2.3.8 Supported Algorithms Extension

The Supported Algorithms extension contains a list of all algorithms
supported by the system. This tells key management what algorithms it
can negotiate. Available authentication algorithms are listed in the
SUPPORTED_AUTH extension and available encryption algorithms are
listed in the SUPPORTED_ENCRYPT extension. The format of these
extensions is:

```
        struct sadb_supported {
                uint16_t sadb_supported_len;
                uint16_t sadb_supported_exttype;
                uint32_t sadb_supported_reserved;
        };
        /* sizeof(struct sadb_supported) == 8 */

        /* followed by:
            struct sadb_alg sadb_algs[(sadb_supported_len *
                sizeof(uint64_t) - sizeof(struct sadb_supported)) /
                sizeof(struct sadb_alg)]; */
```

   This header is followed by one or more algorithm descriptions. An
algorithm description looks like:

```
        struct sadb_alg {
                uint8_t sadb_alg_id;
                uint8_t sadb_alg_ivlen;
                uint16_t sadb_alg_minbits;
                uint16_t sadb_alg_maxbits;
                uint16_t sadb_alg_reserved;
        };
        /* sizeof(struct sadb_alg) == 8 */
```

   sadb_alg_id   The algorithm identification value for this
                   algorithm. This is the value that is stored in
                   sadb_sa_auth or sadb_sa_encrypt if this algorithm is
                   selected.

    sadb_alg_ivlen The length of the initialization vector to be used
                   for the algorithm. If an IV is not needed, this
                   value MUST be set to zero.

    sadb_alg_minbits
                   The minimum acceptable key length, in bits. A value
                   of zero is invalid.

    sadb_alg_maxbits
                   The maximum acceptable key length, in bits. A value
                   of zero is invalid. The minimum MUST NOT be greater
                   than the maximum.

2.3.9 SPI Range Extension

    One PF_KEY message, SADB_GETSPI, might need a range of acceptable SPI
    values. This extension performs such a function.

```
        struct sadb_spirange {
                uint16_t sadb_spirange_len;
                uint16_t sadb_spirange_exttype;
                uint32_t sadb_spirange_min;
                uint32_t sadb_spirange_max;
                uint32_t sadb_spirange_reserved;
        };
        /* sizeof(struct sadb_spirange) == 16 */
```

   sadb_spirange_min
                   The minimum acceptable SPI value.

    sadb_spirange_max
                   The maximum acceptable SPI value. The maximum MUST
                   be greater than or equal to the minimum.

2.4 Illustration of Message Layout

   The following shows how the octets are laid out in a PF_KEY message.
   Optional fields are indicated as such.

   The base header is as follows:

```
   0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
  +---------------+---------------+---------------+---------------+
  |  ...version   | sadb_msg_type | sadb_msg_errno| ...msg_satype |
  +---------------+---------------+---------------+---------------+
  |          sadb_msg_len         |       sadb_msg_reserved       |
  +---------------+---------------+---------------+---------------+
  |                          sadb_msg_seq                         |
  +---------------+---------------+---------------+---------------+
  |                          sadb_msg_pid                         |
  +---------------+---------------+---------------+---------------+
```

   The base header may be followed by one or more of the following
   extension fields, depending on the values of various base header
   fields. The following fields are ordered such that if they appear,
   they SHOULD appear in the order presented below.

   An extension field MUST not be repeated. If there is a situation
   where an extension MUST be repeated, it should be brought to the
   attention of the authors.

   The Association extension

```
       0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
      +---------------+---------------+---------------+---------------+
      |          sadb_sa_len          |         sadb_sa_exttype       |
      +---------------+---------------+---------------+---------------+
      |                          sadb_sa_spi                          |
      +---------------+---------------+---------------+---------------+
      |   ...replay   | sadb_sa_state | sadb_sa_auth  |sadb_sa_encrypt|
      +---------------+---------------+---------------+---------------+
      |                          sadb_sa_flags                        |
      +---------------+---------------+---------------+---------------+
```

   The Lifetime extension

```
      +---------------+---------------+---------------+---------------+
      |         sadb_lifetime_len     |     sadb_lifetime_exttype     |
      +---------------+---------------+---------------+---------------+
      |                     sadb_lifetime_allocations                 |
      +---------------+---------------+---------------+---------------+
```

```
+--------------+--------------+--------------+--------------+
|                    sadb_lifetime_bytes                    |
|                        (64 bits)                          |
+--------------+--------------+--------------+--------------+
|                    sadb_lifetime_addtime                  |
|                        (64 bits)                          |
+--------------+--------------+--------------+--------------+
|                    sadb_lifetime_usetime                  |
|                        (64 bits)                          |
+--------------+--------------+--------------+--------------+
```

The Address extension

```
+--------------+--------------+--------------+--------------+
|         sadb_address_len    |       sadb_address_exttype  |
+--------------+--------------+--------------+--------------+
| _address_proto| ..._prefixlen |     sadb_address_reserved |
+--------------+--------------+--------------+--------------+
>      Some form of 64-bit aligned struct sockaddr goes here.   <
+--------------+--------------+--------------+--------------+
```

The Key extension

```
+--------------+--------------+--------------+--------------+
|          sadb_key_len       |         sadb_key_exttype    |
+--------------+--------------+--------------+--------------+
|          sadb_key_bits      |         sadb_key_reserved   |
+--------------+--------------+--------------+--------------+
>    A key, padded to 64-bits, most significant bits to least. >
+--------------+--------------+--------------+--------------+
```

The Identity extension

```
+--------------+--------------+--------------+--------------+
|         sadb_ident_len      |        sadb_ident_exttype   |
+--------------+--------------+--------------+--------------+
|         sadb_ident_type     |        sadb_ident_reserved  |
+--------------+--------------+--------------+--------------+
|                      sadb_ident_id                        |
|                        (64 bits)                          |
+--------------+--------------+--------------+--------------+
> A null-terminated C-string which MUST be padded out for   >
< 64-bit alignment.                                         <
+--------------+--------------+--------------+--------------+
```

The Sensitivity extension

```
+--------------+--------------+--------------+--------------+
|         sadb_sens_len       |        sadb_sens_exttype     |
+--------------+--------------+--------------+--------------+
|                         sadb_sens_dpd                     |
+--------------+--------------+--------------+--------------+
| ...sens_level | ...sens_len   |..._integ_level| ..integ_len |
+--------------+--------------+--------------+--------------+
|                      sadb_sens_reserved                   |
+--------------+--------------+--------------+--------------+
>     The sensitivity bitmap, followed immediately by the    <
<     integrity bitmap, each is an array of uint64_t.        >
+--------------+--------------+--------------+--------------+
```

The Proposal extension

```
+--------------+--------------+--------------+--------------+
|         sadb_prop_len       |        sadb_prop_exttype     |
+--------------+--------------+--------------+--------------+
|...prop_replay |             sadb_prop_reserved            |
+--------------+--------------+--------------+--------------+
>      One or more combinations, specified as follows...     <
+--------------+--------------+--------------+--------------+
```

Combination

```
+--------------+--------------+--------------+--------------+
|sadb_comb_auth |sadb_comb_encr |        sadb_comb_flags      |
+--------------+--------------+--------------+--------------+
|     sadb_comb_auth_minbits    |     sadb_comb_auth_maxbits   |
+--------------+--------------+--------------+--------------+
|   sadb_comb_encrypt_minbits   |   sadb_comb_encrypt_maxbits  |
+--------------+--------------+--------------+--------------+
|                      sadb_comb_reserved                   |
+--------------+--------------+--------------+--------------+
|                  sadb_comb_soft_allocations               |
+--------------+--------------+--------------+--------------+
|                  sadb_comb_hard_allocations               |
+--------------+--------------+--------------+--------------+
|                   sadb_comb_soft_bytes                    |
|                       (64 bits)                          |
+--------------+--------------+--------------+--------------+
|                   sadb_comb_hard_bytes                    |
|                       (64 bits)                          |
+--------------+--------------+--------------+--------------+
|                   sadb_comb_soft_addtime                  |
|                       (64 bits)                          |
+--------------+--------------+--------------+--------------+
```

```
+--------------+--------------+--------------+--------------+
|                     sadb_comb_hard_addtime                |
|                         (64 bits)                         |
+--------------+--------------+--------------+--------------+
|                     sadb_comb_soft_usetime                |
|                         (64 bits)                         |
+--------------+--------------+--------------+--------------+
|                     sadb_comb_hard_usetime                |
|                         (64 bits)                         |
+--------------+--------------+--------------+--------------+
```

   The Supported Algorithms extension

```
+--------------+--------------+--------------+--------------+
|        sadb_supported_len    |      sadb_supported_exttype |
+--------------+--------------+--------------+--------------+
|                    sadb_supported_reserved                |
+--------------+--------------+--------------+--------------+
```

    Followed by one or more Algorithm Descriptors

```
+--------------+--------------+--------------+--------------+
|  sadb_alg_id | sadb_alg_ivlen|       sadb_alg_minbits     |
+--------------+--------------+--------------+--------------+
|         sadb_alg_maxbits     |       sadb_alg_reserved     |
+--------------+--------------+--------------+--------------+
```

   The SPI Range extension

```
+--------------+--------------+--------------+--------------+
|          sadb_spirange_len   |      sadb_spirange_exttype  |
+--------------+--------------+--------------+--------------+
|                     sadb_spirange_min                     |
+--------------+--------------+--------------+--------------+
|                     sadb_spirange_max                     |
+--------------+--------------+--------------+--------------+
|                   sadb_spirange_reserved                  |
+--------------+--------------+--------------+--------------+
```

3 Symbolic Names

   This section defines various symbols used with PF_KEY and the
   semantics associated with each symbol. Applications MUST use the
   symbolic names in order to be portable. The numeric definitions
   shown are for illustrative purposes, unless explicitly stated
   otherwise. The numeric definition MAY vary on other systems. The
   symbolic name MUST be kept the same for all conforming
   implementations.

3.1 Message Types

   The following message types are used with PF_KEY. These are defined
   in the file <net/pfkeyv2.h>.

```
        #define SADB_RESERVED      0
        #define SADB_GETSPI        1
        #define SADB_UPDATE        2
        #define SADB_ADD           3
        #define SADB_DELETE        4
        #define SADB_GET           5
        #define SADB_ACQUIRE       6
        #define SADB_REGISTER      7
        #define SADB_EXPIRE        8
        #define SADB_FLUSH         9

        #define SADB_DUMP          10   /* not used normally */

        #define SADB_MAX           10
```

   Each message has a behavior. A behavior is defined as where the
   initial message travels (e.g. user to kernel), and what subsequent
   actions are expected to take place. Contents of messages are
   illustrated as:

   <base, REQUIRED EXTENSION, REQ., (OPTIONAL EXT.,) (OPT)>

   The SA extension is sometimes used only for its SPI field. If all
   other fields MUST be ignored, this is represented by "SA(*)".

   The lifetime extensions are represented with one to three letters
   after the word "lifetime," representing (H)ARD, (S)OFT, and
   (C)URRENT.

   The address extensions are represented with one to three letters
   after the word "address," representing (S)RC, (D)ST, (P)ROXY.

      NOTE: Some security association types do not use a source
            address for SA identification, where others do. This may
            cause EEXIST errors for some SA types where others do not
            report collisions. It is expected that application
            authors know enough about the underlying security
            association types to understand these differences.

   The key extensions are represented with one or two letters after the
   word "key," representing (A)UTH and (E)NCRYPT.

The identity extensions are represented with one or two letters after
the word "identity," representing (S)RC and (D)ST.

In the case of an error, only the base header is returned.

Note that any standard error could be returned for any message.

Typically, they will be either one of the errors specifically listed
in the description for a message or one of the following:

        EINVAL Various message improprieties, including SPI ranges
                that are malformed.
        ENOMEM Needed memory was not available.
        ENOBUFS Needed memory was not available.
        EMSGSIZ The message exceeds the maximum length allowed.

3.1.1 SADB_GETSPI

The SADB_GETSPI message allows a process to obtain a unique SPI value
for given security association type, source address, and destination
address. This message followed by an SADB_UPDATE is one way to
create a security association (SADB_ADD is the other method). The
process specifies the type in the base header, the source and
destination address in address extension. If the SADB_GETSPI message
is in response to a kernel-generated SADB_ACQUIRE, the sadb_msg_seq
MUST be the same as the SADB_ACQUIRE message. The application may
also specify the SPI. This is done by having the kernel select
within a range of SPI values by using the SPI range extension. To
specify a single SPI value to be verified, the application sets the
high and low values to be equal. Permitting range specification is
important because the kernel can allocate an SPI value based on what
it knows about SPI values already in use. The kernel returns the
same message with the allocated SPI value stored in the spi field of
an association extension. The allocate SPI (and destination address)
refer to a LARVAL security association. An SADB_UPDATE message can
later be used to add an entry with the requested SPI value.

It is recommended that associations that are created with SADB_GETSPI
SHOULD be automatically deleted within a fixed amount of time if they
are not updated by an SADB_UPDATE message. This allows SA storage
not to get cluttered with larval associations.

    The message behavior of the SADB_GETSPI message is:

        Send an SADB_GETSPI message from a user process to the kernel.

        <base, address, SPI range>

The kernel returns the SADB_GETSPI message to all listening
processes.

<base, SA(*), address(SD)>

Errors:

EEXIST   Requested SPI or SPI range is not available or already
used.

3.1.2 SADB_UPDATE Message

The SADB_UPDATE message allows a process to update the information in
an existing Security Association. Since SADB_GETSPI does not allow
setting of certain parameters, this message is needed to fully form
the SADB_SASTATE_LARVAL security association created with
SADB_GETSPI. The format of the update message is a base header,
followed by an association header and possibly by several extension
headers. The kernel searches for the security association with the
same type, spi, source address and destination address specified in
the message and updates the Security Association information using
the content of the SADB_UPDATE message.

The kernel MAY disallow SADB_UPDATE to succeed unless the message is
issued from the same socket that created the security association.
Such enforcement significantly reduces the chance of accidental
changes to an in-use security association. Malicious trusted parties
could still issue an SADB_FLUSH or SADB_DELETE message, but deletion
of associations is more easily detected and less likely to occur
accidentally than an erroneous SADB_UPDATE. The counter argument to
supporting this behavior involves the case where a user-space key
management application fails and is restarted. The new instance of
the application will not have the same socket as the creator of the
security association.

The kernel MUST sanity check all significant values submitted in an
SADB_UPDATE message before changing the SA in its database and MUST
return EINVAL if any of the values are invalid. Examples of checks
that should be performed are DES key parity bits, key length
checking, checks for keys known to be weak for the specified
algorithm, and checks for flags or parameters known to be
incompatible with the specified algorithm.

Only SADB_SASTATE_MATURE SAs may be submitted in an SADB_UPDATE
message. If the original SA is an SADB_SASTATE_LARVAL SA, then any
value in the SA may be changed except for the source address,
destination address, and SPI. If the original SA is an
SADB_SASTATE_DEAD SA, any attempt to perform an SADB_UPDATE on the SA

   MUST return EINVAL. It is not valid for established keying or
   algorithm information to change without the SPI changing, which would
   require creation of a new SA rather than a change to an existing SA.
   Once keying and algorithm information is negotiated, address and
   identity information is fixed for the SA. Therefore, if the original
   SA is an SADB_SASTATE_MATURE or DYING SA, only the sadb_sa_state
   field in the SA header and lifetimes (hard, soft, and current) may be
   changed and any attempt to change other values MUST result in an
   error return of EINVAL.

   The message behavior of the SADB_UPDATE message is:

      Send an SADB_UPDATE message from a user process to the kernel.

      <base, SA, (lifetime(HSC),) address(SD), (address(P),)
        key(AE), (identity(SD),) (sensitivity)>

      The kernel returns the SADB_UPDATE message to all listening
      processes.

      <base, SA, (lifetime(HSC),) address(SD), (address(P),)
        (identity(SD),) (sensitivity)>

   The keying material is not returned on the message from the kernel to
   listening sockets because listeners might not have the privileges to
   see such keying material.

      Errors:
         ESRCH   The security association to be updated was not found.
         EINVAL In addition to other possible causes, this error is
                 returned if sanity checking on the SA values (such
                 as the keys) fails.
         EACCES Insufficient privilege to update entry. The socket
                 issuing the SADB_UPDATE is not creator of the entry
                     to be updated.

3.1.3 SADB_ADD

   The SADB_ADD message is nearly identical to the SADB_UPDATE message,
   except that it does not require a previous call to SADB_GETSPI. The
   SADB_ADD message is used in manual keying applications, and in other
   cases where the uniqueness of the SPI is known immediately.

   An SADB_ADD message is also used when negotiation is finished, and
   the second of a pair of associations is added. The SPI for this
   association was determined by the peer machine. The sadb_msg_seq

MUST be set to the value set in a kernel-generated SADB_ACQUIRE so
that both associations in a pair are bound to the same ACQUIRE
request.

The kernel MUST sanity check all used fields in the SA submitted in
an SADB_ADD message before adding the SA to its database and MUST
return EINVAL if any of the values are invalid.

Only SADB_SASTATE_MATURE SAs may be submitted in an SADB_ADD message.
SADB_SASTATE_LARVAL SAs are created by SADB_GETSPI and it is not
sensible to add a new SA in the DYING or SADB_SASTATE_DEAD state.
Therefore, the sadb_sa_state field of all submitted SAs MUST be
SADB_SASTATE_MATURE and the kernel MUST return an error if this is
not true.

   The message behavior of the SADB_ADD message is:

      Send an SADB_ADD message from a user process to the kernel.

      <base, SA, (lifetime(HS),) address(SD), (address(P),)
        key(AE), (identity(SD),) (sensitivity)>

      The kernel returns the SADB_ADD message to all listening
      processes.

      <base, SA, (lifetime(HS),) address(SD), (identity(SD),)
        (sensitivity)>

   The keying material is not returned on the message from the kernel to
   listening sockets because listeners may not have the privileges to
   see such keying material.

      Errors:

         EEXIST The security association that was to be added already
                 exists.
         EINVAL In addition to other possible causes, this error is
                 returned if sanity checking on the SA values (such
                 as the keys) fails.

3.1.4 SADB_DELETE

   The SADB_DELETE message causes the kernel to delete a Security
   Association from the key table. The delete message consists of the
   base header followed by the association, and the source and
   destination sockaddrs in the address extension. The kernel deletes
   the security association matching the type, spi, source address, and
   destination address in the message.

    The message behavior for SADB_DELETE is as follows:

        Send an SADB_DELETE message from a user process to the kernel.

        <base, SA(*), address(SD)>

        The kernel returns the SADB_DELETE message to all listening
        processes.

        <base, SA(*), address(SD)>

3.1.5 SADB_GET

   The SADB_GET message allows a process to retrieve a copy of a
   Security Association from the kernel's key table. The get message
   consists of the base header follows by the relevant extension fields.
   The Security Association matching the type, spi, source address, and
   destination address is returned.

     The message behavior of the SADB_GET message is:

        Send an SADB_GET message from a user process to the kernel.

        <base, SA(*), address(SD)>

        The kernel returns the SADB_GET message to the socket that sent
        the SADB_GET message.

        <base, SA, (lifetime(HSC),) address(SD), (address(P),) key(AE),
          (identity(SD),) (sensitivity)>

     Errors:
        ESRCH   The sought security association was not found.

3.1.6 SADB_ACQUIRE

   The SADB_ACQUIRE message is typically sent only by the kernel to key
   socket listeners who have registered their key socket (see
   SADB_REGISTER message). SADB_ACQUIRE messages can be sent by
   application-level consumers of security associations (such as an
   OSPFv2 implementation that uses OSPF security). The SADB_ACQUIRE
   message is a base header along with an address extension, possibly an
   identity extension, and a proposal extension. The proposed situation
   contains a list of desirable algorithms that can be used if the
   algorithms in the base header are not available. The values for the
   fields in the base header and in the security association data which
   follows the base header indicate the properties of the Security
   Association that the listening process should attempt to acquire. If

the message originates from the kernel (i.e. the sadb_msg_pid is 0),
the sadb_msg_seq number MUST be used by a subsequent SADB_GETSPI and
SADB_UPDATE, or subsequent SADB_ADD message to bind a security
association to the request. This avoids the race condition of two
TCP connections between two IP hosts that each require unique
associations, and having one steal another's security association.
The sadb_msg_errno and sadb_msg_state fields should be ignored by the
listening process.

The SADB_ACQUIRE message is typically triggered by an outbound packet
that needs security but for which there is no applicable Security
Association existing in the key table. If the packet can be
sufficiently protected by more than one algorithm or combination of
options, the SADB_ACQUIRE message MUST order the preference of
possibilities in the Proposal extension.

There are three messaging behaviors for SADB_ACQUIRE. The first is
where the kernel needs a security association (e.g. for IPsec).

   The kernel sends an SADB_ACQUIRE message to registered sockets.

      <base, address(SD), (address(P)), (identity(SD),) (sensitivity,)
        proposal>

      NOTE: The address(SD) extensions MUST have the port fields
              filled in with the port numbers of the session requiring
              keys if appropriate.

The second is when, for some reason, key management fails, it can
send an ACQUIRE message with the same sadb_msg_seq as the initial
ACQUIRE with a non-zero errno.

      Send an SADB_ACQUIRE to indicate key management failure.

      <base>

The third is where an application-layer consumer of security
associations (e.g. an OSPFv2 or RIPv2 daemon) needs a security
association.

      Send an SADB_ACQUIRE message from a user process to the kernel.

      <base, address(SD), (address(P),) (identity(SD),) (sensitivity,)
        proposal>

      The kernel returns an SADB_ACQUIRE message to registered
        sockets.

          <base, address(SD), (address(P),) (identity(SD),) (sensitivity,)
            proposal>

          The user-level consumer waits for an SADB_UPDATE or SADB_ADD
          message for its particular type, and then can use that
          association by using SADB_GET messages.

     Errors:
          EINVAL Invalid acquire request.
          EPROTONOSUPPORT   No KM application has registered with the Key
                  Engine as being able to obtain the requested SA type, so
                  the requested SA cannot be acquired.

3.1.7 SADB_REGISTER

   The SADB_REGISTER message allows an application to register its key
   socket as able to acquire new security associations for the kernel.
   SADB_REGISTER allows a socket to receive SADB_ACQUIRE messages for
   the type of security association specified in sadb_msg_satype. The
   application specifies the type of security association that it can
   acquire for the kernel in the type field of its register message. If
   an application can acquire multiple types of security association, it
   MUST register each type in a separate message. Only the base header
   is needed for the register message. Key management applications MAY
   register for a type not known to the kernel, because the consumer may
   be in user-space (e.g. OSPFv2 security).

   The reply of the SADB_REGISTER message contains a supported algorithm
   extension. That field contains an array of supported algorithms, one
   per octet. This allows key management applications to know what
   algorithm are supported by the kernel.

   In an environment where algorithms can be dynamically loaded and
   unloaded, an asynchronous SADB_REGISTER reply MAY be generated. The
   list of supported algorithms MUST be a complete list, so the
   application can make note of omissions or additions.

     The messaging behavior of the SADB_REGISTER message is:

          Send an SADB_REGISTER message from a user process to the kernel.

          <base>

          The kernel returns an SADB_REGISTER message to registered
          sockets, with algorithm types supported by the kernel being
          indicated in the supported algorithms field.

          NOTE: This message may arrive asynchronously due to an
                 algorithm being loaded or unloaded into a dynamically
                 linked kernel.

        <base, supported>

3.1.8 SADB_EXPIRE Message

   The operating system kernel is responsible for tracking SA
   expirations for security protocols that are implemented inside the
   kernel. If the soft limit or hard limit of a Security Association
   has expired for a security protocol implemented inside the kernel,
   then the kernel MUST issue an SADB_EXPIRE message to all key socket
   listeners. If the soft limit or hard limit of a Security Association
   for a user-level security protocol has expired, the user-level
   protocol SHOULD issue an SADB_EXPIRE message.

   The base header will contain the security association information
   followed by the source sockaddr, destination sockaddr, (and, if
   present, internal sockaddr,) (and, if present, one or both
   compartment bitmaps).

   The lifetime extension of an SADB_EXPIRE message is important to
   indicate which lifetime expired. If a HARD lifetime extension is
   included, it indicates that the HARD lifetime expired. This means
   the association MAY be deleted already from the SADB. If a SOFT
   lifetime extension is included, it indicates that the SOFT lifetime
   expired. The CURRENT lifetime extension will indicate the current
   status, and comparisons to the HARD or SOFT lifetime will indicate
   which limit was reached. HARD lifetimes MUST take precedence over
   SOFT lifetimes, meaning if the HARD and SOFT lifetimes are the same,
   the HARD lifetime will appear on the EXPIRE message. The
   pathological case of HARD lifetimes being shorter than SOFT lifetimes
   is handled such that the SOFT lifetime will never expire.

      The messaging behavior of the SADB_EXPIRE message is:

           The kernel sends an SADB_EXPIRE message to all listeners when
           the soft limit of a security association has been expired.

           <base, SA, lifetime(C and one of HS), address(SD)>

   Note that the SADB_EXPIRE message is ONLY sent by the kernel to the
   KMd. It is a one-way informational message that does not have a
   reply.

3.1.9 SADB_FLUSH

   The SADB_FLUSH message causes the kernel to delete all entries in its
   key table for a certain sadb_msg_satype. Only the base header is
   required for a flush message. If sadb_msg_satype is filled in with a
   specific value, only associations of that type are deleted. If it is
   filled in with SADB_SATYPE_UNSPEC, ALL associations are deleted.

      The messaging behavior for SADB_FLUSH is:

           Send an SADB_FLUSH message from a user process to the kernel.

           <base>

           The kernel will return an SADB_FLUSH message to all listening
           sockets.

           <base>

           The reply message happens only after the actual flushing
           of security associations has been attempted.

3.1.10 SADB_DUMP

   The SADB_DUMP message causes the kernel to dump the operating
   system's entire Key Table to the requesting key socket. As in
   SADB_FLUSH, if a sadb_msg_satype value is in the message, only
   associations of that type will be dumped. If SADB_SATYPE_UNSPEC is
   specified, all associations will be dumped. Each Security Association
   is returned in its own SADB_DUMP message. A SADB_DUMP message with a
   sadb_seq field of zero indicates the end of the dump transaction. The
   dump message is used for debugging purposes only and is not intended
   for production use.

   Support for the dump message MAY be discontinued in future versions
   of PF_KEY. Key management applications MUST NOT depend on this
   message for basic operation.

      The messaging behavior for SADB_DUMP is:

           Send an SADB_DUMP message from a user process to the kernel.

           <base>

           Several SADB_DUMP messages will return from the kernel to the
           sending socket.

```
      <base, SA, (lifetime (HSC),) address(SD), (address(P),)
        key(AE), (identity(SD),) (sensitivity)>
```

3.2 Security Association Flags

   The Security Association's flags are a bitmask field. These flags
   also appear in a combination that is part of a PROPOSAL extension.
   The related symbolic definitions below should be used in order that
   applications will be portable:

      #define SADB_SAFLAGS_PFS 1   /* perfect forward secrecy */

   The SADB_SAFLAGS_PFS flag indicates to key management that this
   association should have perfect forward secrecy in its key. (In
   other words, any given session key cannot be determined by
   cryptanalysis of previous session keys or some master key.)

3.3 Security Association States

   The security association state field is an integer that describes the
   states of a security association. They are:

      #define SADB_SASTATE_LARVAL   0
      #define SADB_SASTATE_MATURE   1
      #define SADB_SASTATE_DYING   2
      #define SADB_SASTATE_DEAD   3

      #define SADB_SASTATE_MAX   3

   A SADB_SASTATE_LARVAL security association is one that was created by
   the SADB_GETSPI message. A SADB_SASTATE_MATURE association is one
   that was updated with the SADB_UPDATE message or added with the
   SADB_ADD message. A DYING association is one whose soft lifetime has
   expired. A SADB_SASTATE_DEAD association is one whose hard lifetime
   has expired, but hasn't been reaped by system garbage collection. If
   a consumer of security associations has to extend an association
   beyond its normal lifetime (e.g. OSPF Security) it MUST only set the
   soft lifetime for an association.

3.4 Security Association Types

   This defines the type of Security Association in this message. The
   symbolic names are always the same, even on different
   implementations. Applications SHOULD use the symbolic name in order
   to have maximum portability across different implementations. These
   are defined in the file <net/pfkeyv2.h>.

```
      #define SADB_SATYPE_UNSPEC       0

      #define SADB_SATYPE_AH           2  /* RFC-1826 */
      #define SADB_SATYPE_ESP          3  /* RFC-1827 */

      #define SADB_SATYPE_RSVP         5  /* RSVP Authentication */
      #define SADB_SATYPE_OSPFV2       6  /* OSPFv2 Authentication */
      #define SADB_SATYPE_RIPV2        7  /* RIPv2 Authentication */
      #define SADB_SATYPE_MIP          8  /* Mobile IP Auth. */

      #define SADB_SATYPE_MAX          8
```

   SADB_SATYPE_UNSPEC is defined for completeness and means no specific
   type of security association. This type is never used with PF_KEY
   SAs.

   SADB_SATYPE_AH is for the IP Authentication Header [Atk95b].

   SADB_SATYPE_ESP is for the IP Encapsulating Security Payload
   [Atk95c].

   SADB_SATYPE_RSVP is for the RSVP Integrity Object.

   SADB_SATYPE_OSPFV2 is for OSPFv2 Cryptographic authentication
   [Moy98].

   SADB_SATYPE_RIPV2 is for RIPv2 Cryptographic authentication [BA97].

   SADB_SATYPE_MIP is for Mobile IP's authentication extensions [Per97].

   SADB_SATYPE_MAX is always set to the highest valid numeric value.

3.5 Algorithm Types

   The algorithm type is interpreted in the context of the Security
   Association type defined above. The numeric value might vary between
   implementations, but the symbolic name MUST NOT vary between
   implementations. Applications should use the symbolic name in order
   to have maximum portability to various implementations.

   Some of the algorithm types defined below might not be standardized
   or might be deprecated in the future. To obtain an assignment for a
   symbolic name, contact the authors.

      The symbols below are defined in <net/pfkeyv2.h>.

```
        /* Authentication algorithms */
        #define SADB_AALG_NONE          0
        #define SADB_AALG_MD5HMAC       2
        #define SADB_AALG_SHA1HMAC      3
        #define SADB_AALG_MAX           3

        /* Encryption algorithms */
        #define SADB_EALG_NONE          0
        #define SADB_EALG_DESCBC        2
        #define SADB_EALG_3DESCBC       3
        #define SADB_EALG_NULL          11
        #define SADB_EALG_MAX           11
```

The algorithm for SADB_AALG_MD5_HMAC is defined in [MG98a]. The
algorithm for SADB_AALG_SHA1HMAC is defined in [MG98b]. The
algorithm for SADB_EALG_DESCBC is defined in [MD98]. SADB_EALG_NULL
is the NULL encryption algorithm, defined in [GK98]. The
SADB_EALG_NONE value is not to be used in any security association
except those which have no possible encryption algorithm in them
(e.g. IPsec AH).

3.6 Extension Header Values

   To briefly recap the extension header values:

```
        #define SADB_EXT_RESERVED           0
        #define SADB_EXT_SA                 1
        #define SADB_EXT_LIFETIME_CURRENT   2
        #define SADB_EXT_LIFETIME_HARD      3
        #define SADB_EXT_LIFETIME_SOFT      4
        #define SADB_EXT_ADDRESS_SRC        5
        #define SADB_EXT_ADDRESS_DST        6
        #define SADB_EXT_ADDRESS_PROXY      7
        #define SADB_EXT_KEY_AUTH           8
        #define SADB_EXT_KEY_ENCRYPT        9
        #define SADB_EXT_IDENTITY_SRC       10
        #define SADB_EXT_IDENTITY_DST       11
        #define SADB_EXT_SENSITIVITY        12
        #define SADB_EXT_PROPOSAL           13
        #define SADB_EXT_SUPPORTED_AUTH     14
        #define SADB_EXT_SUPPORTED_ENCRYPT  15
        #define SADB_EXT_SPIRANGE           16

        #define SADB_EXT_MAX                16
```

3.7 Identity Extension Values

   Each identity can have a certain type.

           #define SADB_IDENTTYPE_RESERVED 0
           #define SADB_IDENTTYPE_PREFIX   1
           #define SADB_IDENTTYPE_FQDN   2
           #define SADB_IDENTTYPE_USERFQDN 3

           #define SADB_IDENTTYPE_MAX   3

   The PREFIX identity string consists of a network address followed by a
   forward slash and a prefix length. The network address is in a
   printable numeric form appropriate for the protocol family. The
   prefix length is a decimal number greater than or equal to zero and
   less than the number of bits in the network address. It indicates the
   number of bits in the network address that are significant; all bits
   in the network address that are not significant MUST be set to zero.
   Note that implementations MUST parse the contents of the printable
   address into a binary form for comparison purposes because multiple
   printable strings are valid representations of the same address in
   many protocol families (for example, some allow leading zeros and some
   have letters that are case insensitive). Examples of PREFIX identities
   are "199.33.248.64/27" and "3ffe::1/128". If the source or destination
   identity is a PREFIX identity, the source or destination address for
   the SA (respectively) MUST be within that prefix. The sadb_ident_id
   field is zeroed for these identity types.

   The FQDN identity string contains a fully qualified domain name. An
   example FQDN identity is "ministry-of-truth.inner.net". The
   sadb_ident_id field is zeroed for these identity types.

   The UserFQDN identity consists of a text string in the format commonly
   used for Internet-standard electronic mail. The syntax is the text
   username, followed by the "@" character, followed in turn by the
   appropriate fully qualified domain name. This identity specifies both
   a username and an associated FQDN. There is no requirement that this
   string specify a mailbox valid for SMTP or other electronic mail
   use. This identity is useful with protocols supporting user-oriented
   keying. It is a convenient identity form because the DNS Security
   extensions can be used to distribute signed public key values by
   associating KEY and SIG records with an appropriate MB DNS record. An
   example UserFQDN identity is "julia@ministry-of-love.inner.net". The
   sadb_ident_id field is used to contain a POSIX user id in the absence
   of an identity string itself so that a user-level application can use
   the getpwuid{,_r}() routine to obtain a textual user login id. If a
   string is present, it SHOULD match the numeric value in the
   sadb_ident_id field. If it does not match, the string SHOULD override

the numeric value.

3.8 Sensitivity Extension Values

   The only field currently defined in the sensitivity extension is the
   sadb_sens_dpd, which represents the data protection domain. The other
   data in the sensitivity extension is based off the sadb_sens_dpd
   value.

   The DP/DOI is defined to be the same as the "Labeled Domain Identifier
   Value" of the IP Security DOI specification [Pip98]. As noted in that
   specification, values in the range 0x80000000 to 0xffffffff
   (inclusive) are reserved for private use and values in the range
   0x00000001 through 0x7fffffff are assigned by IANA. The all-zeros
   DP/DOI value is permanently reserved to mean that "no DP/DOI is in
   use".

3.9 Proposal Extension Values

   These are already mentioned in the Algorithm Types and Security
   Association Flags sections.

4 Future Directions

   While the current specification for the Sensitivity and Integrity
   Labels is believed to be general enough, if a case should arise that
   can't work with the current specification then this might cause a
   change in a future version of PF_KEY.

   Similarly, PF_KEY might need extensions to work with other kinds of
   Security Associations in future. It is strongly desirable for such
   extensions to be made in a backwards-compatible manner should they be
   needed.

   When more experience is gained with certificate management, it is
   possible that the IDENTITY extension will have to be revisited to
   allow a finer grained selection of certificate identities.

5. Examples

   The following examples illustrate how PF_KEY is used. The first
   example is an IP Security example, where the consumer of the security
   associations is inside an operating system kernel. The second example
   is an OSPF Security example, which illustrates a user-level consumer
   of security associations. The third example covers things not
   mentioned by the first two examples. A real system may closely
   conform to one of these examples, or take parts of them. These
   examples are purely illustrative, and are not intended to mandate a

   particular implementation method.

5.1 Simple IP Security Example

```
              +---------------+     +-------------+
              |Key Mgmt Daemon|     | Application |
              +---------------+     +-------------+
                 |       |      /
                 |       |     /
                 |       |    |            Applications
          ======[PF_KEY]====[PF_INET]=========================
                 |       |    |            OS Kernel
           +-----------+   +----------------+
           | Key Engine |   | TCP/IP,        |
           |  or  SADB  |---| including IPsec |
           +-----------+   |                |
                           +----------------+
```
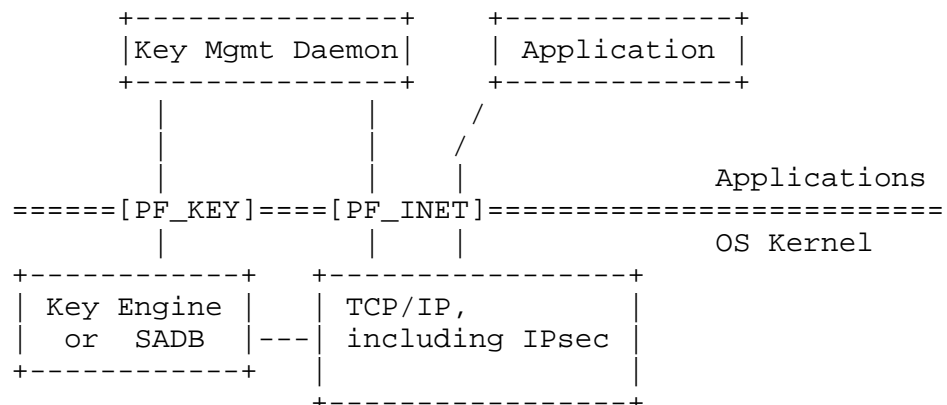
   When the Key Management daemon (KMd) begins. It must tell PF_KEY
   that it is willing to accept message for the two IPsec services, AH
   and ESP. It does this by sending down two SADB_REGISTER messages.

      KMd->Kernel: SADB_REGISTER for ESP
      Kernel->Registered: SADB_REGISTER for ESP, Supported Algorithms
      KMd->Kernel: SADB_REGISTER for AH
      Kernel->Registered: SADB_REGISTER for AH, Supported Algorithms

   Each REGISTER message will cause a reply to go to all PF_KEY sockets
   registered for ESP and AH respectively (including the requester).

   Assume that no security associations currently exist for IPsec to
   use. Consider when a network application begins transmitting data
   (e.g. a TCP SYN). Because of policy, or the application's request,
   the kernel IPsec module needs an AH security association for this
   data. Since there is not one present, the following message is
   generated:

      Kernel->Registered: SADB_ACQUIRE for AH, addrs, ID, sens,
                           proposals

   The KMd reads the ACQUIRE message, especially the sadb_msg_seq
   number. Before it begins the negotiation, it sends down an
   SADB_GETSPI message with the sadb_msg_seq number equal to the one
   received in the ACQUIRE. The kernel returns the results of the
   GETSPI to all listening sockets.

      KMd->Kernel: SADB_GETSPI for AH, addr, SPI range
      Kernel->All: SADB_GETSPI for AH, assoc, addrs

The KMd may perform a second GETSPI operation if it needs both
directions of IPsec SPI values. Now that the KMd has an SPI for at
least one of the security associations, it begins negotiation. After
deriving keying material, and negotiating other parameters, it sends
down one (or more) SADB_UPDATE messages with the same value in
sadb_msg_seq.

If a KMd has any error at all during its negotiation, it can send
down:

    KMd->Kernel: SADB_ACQUIRE for AH, assoc (with an error)
    Kernel->All: SADB_ACQUIRE for AH, assoc (same error)

but if it succeeds, it can instead:

    KMd->Kernel: SADB_UPDATE for AH, assoc, addrs, keys,
                           <etc.>
    Kernel->All: SADB_UPDATE for AH, assoc, addrs, <etc.>

The results of the UPDATE (minus the actual keys) are sent to all
listening sockets. If only one SPI value was determined locally, the
other SPI (since IPsec SAs are unidirectional) must be added with an
SADB_ADD message.

    KMd->Kernel: SADB_ADD for AH, assoc, addrs, keys, <etc.>
    Kernel->All: SADB_ADD for AH, assoc, addrs, <etc.>

If one of the extensions passed down was a Lifetime extension, it is
possible at some point an SADB_EXPIRE message will arrive when one of
the lifetimes has expired.

    Kernel->All: SADB_EXPIRE for AH, assoc, addrs,
                           Hard or Soft, Current, <etc.>

The KMd can use this as a clue to begin negotiation, or, if it has
some say in policy, send an SADB_UPDATE down with a lifetime
extension.

5.2 Proxy IP Security Example

Many people are interested in using IP Security in a "proxy" or
"firewall" configuration in which an intermediate system provides
security services for "inside" hosts. In these environments, the
intermediate systems can use PF_KEY to communicate with key
management applications almost exactly as they would if they were the
actual endpoints. The messaging behavior of PF_KEY in these cases is
exactly the same as the previous example, but the address information
is slightly different.

Consider this case:

                    A ========= B --------- C

    Key:
                A   "outside" host that implements IPsec
                B   "firewall" that implements IPsec
                C   "inside" host that does not implement IPsec

                ===   IP_{A<->B} ESP [IP_{A<->C} ULP]
                ---   IP_{A<->C} ULP

A is a single system that wishes to communicate with the "inside"
system C. B is a "firewall" between C and the outside world that
will do ESP and tunneling on C's behalf. A discovers that it needs
to send traffic to C via B through methods not described here (Use of
the DNS' KX record might be one method for discovering this).

For packets that flow from left to right, A and B need an IPsec
Security Association with:

        SA type of ESP tunnel-mode
        Source Identity that dominates A (e.g. A's address)
        Destination Identity that dominates B (e.g. B's address)
        Source Address of A
        Destination Address of B

For packets to flow from right to left, A and B need an IPsec
Security Association with:

        SA type of ESP tunnel-mode
        Source Identity that dominates C
        Destination Identity that dominates A
        Source Address of B
        Destination Address of A
        Proxy Address of C

For this second SA (for packets flowing from C towards A), node A
MUST verify that the inner source address is dominated by the Source
Identity for the SA used with those packets. If node A does not do
this, an adversary could forge packets with an arbitrary Source
Identity and defeat the packet origin protections provided by IPsec.

    Now consider a slightly more complex case:

            A_1 --| |-- D_1
                  |--- B ====== C ---|
            A_2 --| |-- D_2

```
     Key:
                A_n      "inside" host on net 1 that does not do IPsec.
                B        "firewall" for net 1 that supports IPsec.
                C        "firewall" for net 2 that supports IPsec.
                D_n      "inside" host on net 2 that does not do IPsec.
                ===      IP_{B<->C} ESP [ IP_{A<->C} ULP ]
                ---      IP_{A<->C} ULP
```

For A_1 to send a packet to D_1, B and C need an SA with:

```
        SA Type of ESP
        Source Identity that dominates A_1
        Destination Identity that dominates C
        Source Address of B
        Destination Address of C
        Proxy Address of A_1
```

For D_1 to send a packet to A_1, C and B need an SA with:
```
        SA Type of ESP Tunnel-mode
        Source Identity that dominates D_1
        Destination Identity that dominates B
        Source Address of C
        Destination Address of B
        Proxy Address of D_1
```

Note that A_2 and D_2 could be substituted for A_1 and D_1
(respectively) here; the association of an SA with a particular pair
of ends or group of those pairs is a policy decision on B and/or C
and not necessarily a function of key management. The same check of
the Source Identity against the inner source IP address MUST also be
performed in this case for the same reason.

For a more detailed discussion of the use of IP Security in complex
cases, please see [Atk97].

  NOTE: The notion of identity domination might be unfamiliar. Let H
  represent some node. Let Hn represent H's fully qualified domain
  name. Let Ha represent the IP address of H. Let Hs represent the IP
  subnet containing Ha. Let Hd represent a fully qualified domain
  name that is a parent of the fully qualified domain name of H. Let
  M be a UserFQDN identity that whose right-hand part is Hn or Ha.

  Any of M, Hn, Ha, Hs, and Hd is considered to dominate H in the
  example above. Hs dominates any node having an IP address within
  the IP address range represented by Hs. Hd dominates any node
  having a fully qualified domain name within underneath Hd.

5.3 OSPF Security Example

```
        +---------------+    +-------------+
        |Key Mgmt Daemon|    | OSPF daemon |
        +---------------+    +-------------+
           |        |    /    /        |
           |  /------|----+   /         |
           | /       |  +---+          |         Applications
      ======[PF_KEY]====[PF_INET]==========[PF_ROUTE]===============
           |        |  |           |         OS Kernel
      +-----------+  +---------------+  +---------+
      | Key Engine |  | TCP/IP,       |  | Routing |
      |  or  SADB  |---| including IPsec |--| Table   |
      +-----------+  |               |  +---------+
                     +---------------+
```
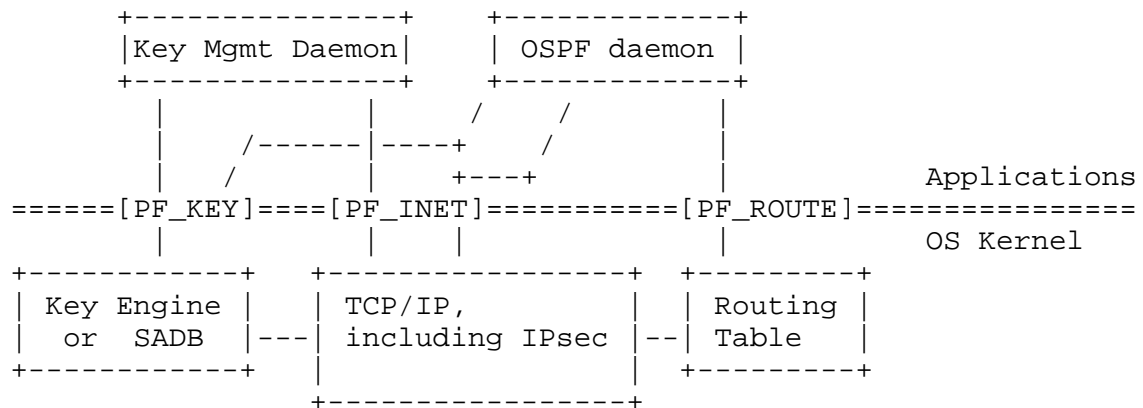
As in the previous examples, the KMd registers itself with the Key
Engine via PF_KEY. Even though the consumer of the security
associations is in user-space, the PF_KEY and Key Engine
implementation knows enough to store SAs and to relay messages.

When the OSPF daemon needs to communicate securely with its peers, it
would perform an SADB_GET message and retrieve the appropriate
association:

```
   OSPFd->Kernel:        SADB_GET of OSPF, assoc, addrs
   Kernel->OSPFd:        SADB_GET of OSPF, assoc, addrs, keys, <etc.>
```

If this GET fails, the OSPFd may need to acquire a new security
association. This interaction is as follows:

```
   OSPFd->Kernel:        SADB_ACQUIRE of OSPF, addrs, <ID, sens,>
                         proposal
   Kernel->Registered:  SADB_ACQUIRE of OSPF, <same as sent message>
```

The KMd sees this and performs actions similar to the previous
example. One difference, however, is that when the UPDATE message
comes back, the OSPFd will then perform a GET of the updated SA to
retrieve all of its parameters.

5.4 Miscellaneous

Some messages work well only in system maintenance programs, for
debugging, or for auditing. In a system panic situation, such as a
detected compromise, an SADB_FLUSH message should be issued for a
particular SA type, or for ALL SA types.

```
   Program->Kernel:       SADB_FLUSH for ALL
   <Kernel then flushes all internal SAs>
   Kernel->All:           SADB_FLUSH for ALL
```

   Some SAs may need to be explicitly deleted, either by a KMd, or by a
   system maintenance program.

```
   Program->Kernel:       SADB_DELETE for AH, association, addrs
   Kernel->All:           SADB_DELETE for AH, association, addrs
```

   Common usage of the SADB_DUMP message is discouraged. For debugging
   purposes, however, it can be quite useful. The output of a DUMP
   message should be read quickly, in order to avoid socket buffer
   overflows.

```
   Program->Kernel:       SADB_DUMP for ESP
   Kernel->Program:       SADB_DUMP for ESP, association, <all fields>
   Kernel->Program:       SADB_DUMP for ESP, association, <all fields>
   Kernel->Program:       SADB_DUMP for ESP, association, <all fields>
   <ad nauseam...>
```

6 Security Considerations

   This memo discusses a method for creating, reading, modifying, and
   deleting Security Associations from an operating system. Only
   trusted, privileged users and processes should be able to perform any
   of these operations. It is unclear whether this mechanism provides
   any security when used with operating systems not having the concept
   of a trusted, privileged user.

   If an unprivileged user is able to perform any of these operations,
   then the operating system cannot actually provide the related
   security services. If an adversary knows the keys and algorithms in
   use, then cryptography cannot provide any form of protection.

   This mechanism is not a panacea, but it does provide an important
   operating system component that can be useful in creating a secure
   internetwork.

   Users need to understand that the quality of the security provided by
   an implementation of this specification depends completely upon the
   overall security of the operating system, the correctness of the
   PF_KEY implementation, and upon the security and correctness of the
   applications that connect to PF_KEY. It is appropriate to use high
   assurance development techniques when implementing PF_KEY and the
   related security association components of the operating system.

Acknowledgments

References

   [AMPMC96] Randall J. Atkinson, Daniel L. McDonald, Bao G. Phan, Craig
   W. Metz, and Kenneth C. Chin, "Implementation of IPv6 in 4.4-Lite
   BSD", Proceedings of the 1996 USENIX Conference, San Diego, CA,
   January 1996, USENIX Association.

   [Atk95a] Atkinson, R., "IP Security Architecture", RFC 1825, August
   1995.

   [Atk95b] Atkinson, R., "IP Authentication Header", RFC 1826, August
   1995.

   [Atk95c] Atkinson, R., "IP Encapsulating Security Payload", RFC 1827,
   August 1995.

   [Atk97] Atkinson, R., "Key Exchange Delegation Record for the Domain
   Name System", RFC 2230, October 1997.

   [BA97] Baker, F., and R. Atkinson, "RIP-2 MD5 Authentication", RFC
   2082, January 1997.

   [Biba77] K. J. Biba, "Integrity Considerations for Secure Computer
   Systems", MTR-3153, The MITRE Corporation, June 1975; ESD-TR-76-372,
   April 1977.

[BL74] D. Elliot Bell and Leonard J. LaPadula, "Secure Computer
Systems: Unified Exposition and Multics Interpretation", MTR 2997,
The MITRE Corporation, April 1974. (AD/A 020 445)

[Bra97] Bradner, S., "Key words for use in RFCs to Indicate
Requirement Levels", BCP 14, RFC 2119, March 1997.

[CW87] D. D. Clark and D. R. Wilson, "A Comparison of Commercial and
Military Computer Security Policies", Proceedings of the 1987
Symposium on Security and Privacy, pp. 184-195, IEEE Computer
Society, Washington, D.C., 1987.

[DIA] US Defense Intelligence Agency (DIA), "Compartmented Mode
Workstation Specification", Technical Report DDS-2600-6243-87.

[GK98] Glenn, R., and S. Kent, "The NULL Encryption Algorithm and Its
Use with IPsec", Work in Progress.

[HM97a] Harney, H., and C. Muckenhirn, "Group Key Management Protocol
(GKMP) Specification", RFC 2093, July 1997.

[HM97b] Harney, H., and C. Muckenhirn, "Group Key Management Protocol
(GKMP) Architecture", RFC 2094, July 1997.

[MD98] Madsen, C., and N. Doraswamy, "The ESP DES-CBC Cipher
Algorithm With Explicit IV", Work in Progress.

[MG98a] Madsen, C., and R. Glenn, "The Use of HMAC-MD5-96 within ESP
and AH", Work in Progress.

[MG98b] Madsen, C., and R. Glenn, "The Use of HMAC-SHA-1-96 within
ESP and AH", Work in Progress.

[MSST98] Maughan, D., Schertler, M., Schneider, M., and J. Turner,
"Internet Security Association and Key Management Protocol (ISAKMP)",
Work in Progress.

[Moy98] Moy, J., "OSPF Version 2", STD 54, RFC 2328, April 1998.

[Per97] Perkins, C., "IP Mobility Support", RFC 2002, October 1996.

[Pip98] Piper, D., "The Internet IP Security Domain of Interpretation
for ISAKMP", Work in Progress.

[Sch96] Bruce Schneier, Applied Cryptography, p. 360, John Wiley &
Sons, Inc., 1996.

   [Skl91] Keith Sklower, "A Tree-based Packet Routing Table for
   Berkeley UNIX", Proceedings of the Winter 1991 USENIX Conference,
   Dallas, TX, USENIX Association. 1991. pp. 93-103.

Disclaimer

   The views and specification here are those of the editors and are not
   necessarily those of their employers. The employers have not passed
   judgment on the merits, if any, of this work. The editors and their
   employers specifically disclaim responsibility for any problems
   arising from correct or incorrect implementation or use of this
   specification.

Authors' Addresses

   Daniel L. McDonald
   Sun Microsystems, Inc.
   901 San Antonio Road, MS UMPK17-202
   Palo Alto, CA 94303

   Phone: +1 650 786 6815
   EMail: danmcd@eng.sun.com

   Craig Metz
   (for Code 5544)
   U.S. Naval Research Laboratory
   4555 Overlook Ave. SW
   Washington, DC 20375

   Phone: (DSN) 754-8590
   EMail: cmetz@inner.net

   Bao G. Phan
   U. S. Naval Research Laboratory

   EMail: phan@itd.nrl.navy.mil

Appendix A: Promiscuous Send/Receive Message Type

   A kernel supporting PF_KEY MAY implement the following extension for
   development and debugging purposes. If it does, it MUST implement the
   extension as specified here. An implementation MAY require an
   application to have additional privileges to perform promiscuous send
   and/or receive operations.

   The SADB_X_PROMISC message allows an application to send and receive
   messages in a "promiscuous mode." There are two forms of this
   message: control and data. The control form consists of only a
   message header. This message is used to toggle the promiscuous-
   receive function. A value of one in the sadb_msg_satype field enables
   promiscuous message reception for this socket, while a value of zero
   in that field disables it.

   The second form of this message is the data form. This is used to
   send or receive messages in their raw form. Messages in the data form
   consist of a message header followed by an entire new message. There
   will be two message headers in a row: one for the SADB_X_PROMISC
   message, and one for the payload message.

   Data messages sent from the application are sent to either the PF_KEY
   socket of a single process identified by a nonzero sadb_msg_seq or to
   all PF_KEY sockets if sadb_msg_seq is zero. These messages are sent
   without any processing of their contents by the PF_KEY interface
   (including sanity checking). This promiscuous-send capability allows
   an application to send messages as if it were the kernel. This also
   allows it to send erroneous messages.

   If the promiscuous-receive function has been enabled, a copy of any
   message sent via PF_KEY by another application or by the kernel is
   sent to the promiscuous application. This is done before any
   processing of the message's contents by the PF_KEY interface (again,
   including sanity checking). This promiscuous-receive capability
   allows an application to receive all messages sent by other parties
   using PF_KEY.

      The messaging behavior of the SADB_X_PROMISC message is:

          Send a control-form SADB_X_PROMISC message from a user process
          to the kernel.

          <base>

          The kernel returns the SADB_X_PROMISC message to all listening
          processes.

        <base>

        Send a data-form SADB_X_PROMISC message from a user process to
        the kernel.

        <base, base(, others)>

        The kernel sends the encapsulated message to the target
        process(s).

        <base(, others)>

        If promiscuous-receive is enabled, the kernel will encapsulate
        and send copies of all messages sent via the PF_KEY interface.

        <base, base(, others)>

    Errors:
        EPERM Additional privileges are required to perform the
             requested operations.
        ESRCH (Data form, sending) The target process in sadb_msg_seq
             does not exist or does not have an open PF_KEY Version 2
             socket.

Appendix B: Passive Change Message Type

   The SADB_X_PCHANGE message is a passive-side (aka. the "listener" or
   "receiver") counterpart to the SADB_ACQUIRE message. It is useful
   for when key management applications wish to more effectively handle
   incoming key management requests for passive-side sessions that
   deviate from systemwide default security services. If a passive
   session requests that only certain levels of security service be
   allowed, the SADB_X_PCHANGE message expresses this change to any
   registered PF_KEY sockets. Unlike SADB_ACQUIRE, this message is
   purely informational, and demands no other PF_KEY interaction.

   The SADB_X_PCHANGE message is typically triggered by either a change
   in an endpoint's requested security services, or when an endpoint
   that made a special request disappears. In the former case, an
   SADB_X_PCHANGE looks like an SADB_ACQUIRE, complete with an
   sadb_proposal extension indicating the preferred algorithms,
   lifetimes, and other attributes. When a passive session either
   disappears, or reverts to a default behavior, an SADB_X_PCHANGE will
   be issued with _no_ sadb_proposal extension, indicating that the
   exception to systemwide default behavior has disappeared.

   There are two messaging behaviors for SADB_X_PCHANGE. The first is
   the kernel-originated case:

        The kernel sends an SADB_X_PCHANGE message to registered
        sockets.

        <base, address(SD), (identity(SD),) (sensitivity,) (proposal)>

        NOTE: The address(SD) extensions MUST have the port fields
               filled in with the port numbers of the session
               requiring keys if appropriate.

   The second is for a user-level consumer of SAs.

        Send an SADB_X_PCHANGE message from a user process to the
        kernel.

        <base, address(SD), (identity(SD),) (sensitivity,) (proposal)>

        The kernel returns an SADB_X_PCHANGE message to registered
        sockets.

        <base, address(SD), (identity(SD),) (sensitivity,) (proposal)>

Appendix C: Key Management Private Data Extension

   The Key Management Private Data extension is attached to either an
   SADB_ADD or an SADB_UPDATE message. It attaches a single piece of
   arbitrary data to a security association. It may be useful for key
   managment applications that could use an SADB_DUMP or SADB_GET
   message to obtain additional state if it needs to restart or recover
   after a crash. The format of this extension is:

           #define SADB_X_EXT_KMPRIVATE 17

           struct sadb_x_kmprivate {
                   uint16_t sadb_x_kmprivate_len;
                   uint16_t sadb_x_kmprivate_exttype;
                   uint32_t sadb_x_kmprivate_reserved;
           };
           /* sizeof(struct sadb_x_kmprivate) == 8 */

           /* followed by arbitrary data */

   The data following the sadb_x_kmprivate extension can be anything.
   It will be stored with the actual security association in the kernel.
   Like all data, it must be padded to an eight byte boundary.

Appendix D: Sample Header File

```
     /*
     This file defines structures and symbols for the PF_KEY Version 2
     key management interface. It was written at the U.S. Naval Research
     Laboratory. This file is in the public domain. The authors ask that
     you leave this credit intact on any copies of this file.
     */
     #ifndef __PFKEY_V2_H
     #define __PFKEY_V2_H 1

     #define PF_KEY_V2 2
     #define PFKEYV2_REVISION        199806L

     #define SADB_RESERVED    0
     #define SADB_GETSPI      1
     #define SADB_UPDATE      2
     #define SADB_ADD         3
     #define SADB_DELETE      4
     #define SADB_GET         5
     #define SADB_ACQUIRE     6
     #define SADB_REGISTER    7
     #define SADB_EXPIRE      8
     #define SADB_FLUSH       9
     #define SADB_DUMP        10
     #define SADB_X_PROMISC   11
     #define SADB_X_PCHANGE   12
     #define SADB_MAX         12

     struct sadb_msg {
       uint8_t sadb_msg_version;
       uint8_t sadb_msg_type;
       uint8_t sadb_msg_errno;
       uint8_t sadb_msg_satype;
       uint16_t sadb_msg_len;
       uint16_t sadb_msg_reserved;
       uint32_t sadb_msg_seq;
       uint32_t sadb_msg_pid;
     };

     struct sadb_ext {
       uint16_t sadb_ext_len;
       uint16_t sadb_ext_type;
     };

     struct sadb_sa {
       uint16_t sadb_sa_len;
       uint16_t sadb_sa_exttype;
```

```
      uint32_t sadb_sa_spi;
      uint8_t sadb_sa_replay;
      uint8_t sadb_sa_state;
      uint8_t sadb_sa_auth;
      uint8_t sadb_sa_encrypt;
      uint32_t sadb_sa_flags;
    };

    struct sadb_lifetime {
      uint16_t sadb_lifetime_len;
      uint16_t sadb_lifetime_exttype;
      uint32_t sadb_lifetime_allocations;
      uint64_t sadb_lifetime_bytes;
      uint64_t sadb_lifetime_addtime;
      uint64_t sadb_lifetime_usetime;
    };

    struct sadb_address {
      uint16_t sadb_address_len;
      uint16_t sadb_address_exttype;
      uint8_t sadb_address_proto;
      uint8_t sadb_address_prefixlen;
      uint16_t sadb_address_reserved;
    };

    struct sadb_key {
      uint16_t sadb_key_len;
      uint16_t sadb_key_exttype;
      uint16_t sadb_key_bits;
      uint16_t sadb_key_reserved;
    };

    struct sadb_ident {
      uint16_t sadb_ident_len;
      uint16_t sadb_ident_exttype;
      uint16_t sadb_ident_type;
      uint16_t sadb_ident_reserved;
      uint64_t sadb_ident_id;
    };

    struct sadb_sens {
      uint16_t sadb_sens_len;
      uint16_t sadb_sens_exttype;
      uint32_t sadb_sens_dpd;
      uint8_t sadb_sens_sens_level;
      uint8_t sadb_sens_sens_len;
      uint8_t sadb_sens_integ_level;
      uint8_t sadb_sens_integ_len;
```

```
      uint32_t sadb_sens_reserved;
    };

    struct sadb_prop {
      uint16_t sadb_prop_len;
      uint16_t sadb_prop_exttype;
      uint8_t sadb_prop_replay;
      uint8_t sadb_prop_reserved[3];
    };

    struct sadb_comb {
      uint8_t sadb_comb_auth;
      uint8_t sadb_comb_encrypt;
      uint16_t sadb_comb_flags;
      uint16_t sadb_comb_auth_minbits;
      uint16_t sadb_comb_auth_maxbits;
      uint16_t sadb_comb_encrypt_minbits;
      uint16_t sadb_comb_encrypt_maxbits;
      uint32_t sadb_comb_reserved;
      uint32_t sadb_comb_soft_allocations;
      uint32_t sadb_comb_hard_allocations;
      uint64_t sadb_comb_soft_bytes;
      uint64_t sadb_comb_hard_bytes;
      uint64_t sadb_comb_soft_addtime;
      uint64_t sadb_comb_hard_addtime;
      uint64_t sadb_comb_soft_usetime;
      uint64_t sadb_comb_hard_usetime;
    };

    struct sadb_supported {
      uint16_t sadb_supported_len;
      uint16_t sadb_supported_exttype;
      uint32_t sadb_supported_reserved;
    };

    struct sadb_alg {
      uint8_t sadb_alg_id;
      uint8_t sadb_alg_ivlen;
      uint16_t sadb_alg_minbits;
      uint16_t sadb_alg_maxbits;
      uint16_t sadb_alg_reserved;
    };

    struct sadb_spirange {
      uint16_t sadb_spirange_len;
      uint16_t sadb_spirange_exttype;
      uint32_t sadb_spirange_min;
      uint32_t sadb_spirange_max;
```

```
      uint32_t sadb_spirange_reserved;
    };

    struct sadb_x_kmprivate {
      uint16_t sadb_x_kmprivate_len;
      uint16_t sadb_x_kmprivate_exttype;
      uint32_t sadb_x_kmprivate_reserved;
    };

    #define SADB_EXT_RESERVED               0
    #define SADB_EXT_SA                     1
    #define SADB_EXT_LIFETIME_CURRENT       2
    #define SADB_EXT_LIFETIME_HARD          3
    #define SADB_EXT_LIFETIME_SOFT          4
    #define SADB_EXT_ADDRESS_SRC            5
    #define SADB_EXT_ADDRESS_DST            6
    #define SADB_EXT_ADDRESS_PROXY          7
    #define SADB_EXT_KEY_AUTH               8
    #define SADB_EXT_KEY_ENCRYPT            9
    #define SADB_EXT_IDENTITY_SRC           10
    #define SADB_EXT_IDENTITY_DST           11
    #define SADB_EXT_SENSITIVITY            12
    #define SADB_EXT_PROPOSAL               13
    #define SADB_EXT_SUPPORTED_AUTH         14
    #define SADB_EXT_SUPPORTED_ENCRYPT      15
    #define SADB_EXT_SPIRANGE               16
    #define SADB_X_EXT_KMPRIVATE            17
    #define SADB_EXT_MAX                    17
    #define SADB_SATYPE_UNSPEC      0
    #define SADB_SATYPE_AH          2
    #define SADB_SATYPE_ESP         3
    #define SADB_SATYPE_RSVP        5
    #define SADB_SATYPE_OSPFV2      6
    #define SADB_SATYPE_RIPV2       7
    #define SADB_SATYPE_MIP         8
    #define SADB_SATYPE_MAX         8

    #define SADB_SASTATE_LARVAL     0
    #define SADB_SASTATE_MATURE     1
    #define SADB_SASTATE_DYING      2
    #define SADB_SASTATE_DEAD       3
    #define SADB_SASTATE_MAX        3

    #define SADB_SAFLAGS_PFS        1

    #define SADB_AALG_NONE          0
    #define SADB_AALG_MD5HMAC       2
    #define SADB_AALG_SHA1HMAC      3
```

```
#define SADB_AALG_MAX          3

#define SADB_EALG_NONE         0
#define SADB_EALG_DESCBC       2
#define SADB_EALG_3DESCBC      3
#define SADB_EALG_NULL         11
#define SADB_EALG_MAX          11

#define SADB_IDENTTYPE_RESERVED   0
#define SADB_IDENTTYPE_PREFIX     1
#define SADB_IDENTTYPE_FQDN       2
#define SADB_IDENTTYPE_USERFQDN   3
#define SADB_IDENTTYPE_MAX        3

#define SADB_KEY_FLAGS_MAX 0
#endif /* __PFKEY_V2_H */
```

Appendix E: Change Log

   The following changes were made between 05 and 06:

   * Last change before becoming an informational RFC. Removed all
     Internet-Draft references. Also standardized citation strings.
     Now cite RFC 2119 for MUST, etc.

   * New appendix on optional KM private data extension.

   * Fixed example to indicate the ACQUIRE messages with errno mean
     KM failure.

   * Added SADB_EALG_NULL.

   * Clarified proxy examples to match definition of PROXY address being
     the inner packet's source address. (Basically a sign-flip. The
     example still shows how to protect against policy vulnerabilities
     in tunnel endpoints.)

   * Loosened definition of a destination address to include broadcast.

   * Recommended that LARVAL security associations have implicit short
     lifetimes.

   The following changes were made between 04 and 05:

   * New appendix on Passive Change message.

   * New sadb_address_prefixlen field.

   * Small clarifications on sadb_ident_id usage.

   * New PFKEYV2_REVISION value.

   * Small clarification on what a PROXY address is.

   * Corrected sadb_spirange_{min,max} language.

   * In ADD messages that are in response to an ACQUIRE, the
     sadb_msg_seq MUST be the same as that of the originating ACQUIRE.

   * Corrected ACQUIRE message behavior, ACQUIRE message SHOULD send up
     PROXY addresses when it needs them.

   * Clarification on SADB_EXPIRE and user-level security protocols.

   The following changes were made between 03 and 04:

* Stronger language about manual keying.

* PFKEYV2_REVISION, ala POSIX.

* Put in language about sockaddr ports in ACQUIRE messages.

* Mention of asymmetric algorithms.

* New sadb_ident_id field for easier construction of USER_FQDN
  identity strings.

* Caveat about source addresses not always used for collision
  detection. (e.g. IPsec)

The following changes were made between 02 and 03:

* Formatting changes.

* Many editorial cleanups, rewordings, clarifications.

* Restrictions that prevent many strange and invalid cases.

* Added definitions section.

* Removed connection identity type (this will reappear when it is
  more clear what it should look like).

* Removed 5.2.1 (Why involve the kernel?).

* Removed INBOUND, OUTBOUND, and FORWARD flags; they can be computed
  from src, dst, and proxy and you had to anyway for sanity checking.

* Removed REPLAY flag; sadb_sa_replay==0 means the same thing.

* Renamed bit lengths to "bits" to avoid potential confusion.

* Explicitly listed lengths for structures.

* Reworked identities to always use a string format.

* Removed requirements for support of shutdown() and SO_USELOOPBACK.

* 64 bit alignment and 64 bit lengths instead of 32 bit.

* time_t replaced with uint64 in lifetimes.

   * Inserted Appendix A (SADB_X_PROMISC) and Appendix B (SAMPLE HEADER
     FILE).

   * Explicit error if PF_KEY_V2 not set at socket() call.

   * More text on SO_USELOOPBACK.

   * Made fields names and symbol names more consistent.

   * Explicit error if PF_KEY_V2 is not in sadb_msg_version field.

   * Bytes lifetime field now a 64-bit quantity.

   * Explicit len/exttype wording.

   * Flattening out of extensions (LIFETIME_HARD, LIFETIME_SOFT, etc.)

   * UI example (0x123 == 0x1230 or 0x0123).

   * Cleaned up and fixed some message behavior examples.

   The following changes were made between 01 and 02:

   * Mentioned that people COULD use these same messages between user
     progs. (Also mentioned why you still might want to use the actual
     socket.)

   * Various wordsmithing changes.

   * Took out netkey/ directory, and make net/pfkeyv2.h

   * Inserted PF_KEY_V2 proto argument per C. Metz.

   * Mentioned other socket calls and how their PF_KEY behavior is
     undefined.

   * SADB_EXPIRE now communicates both hard and soft lifetime expires.

   * New "association" extension, even smaller base header.

   * Lifetime extension improvements.

   * Length now first in extensions.

   * Errors can be sent from kernel to user, also.

   * Examples section inserted.

   * Some bitfield cleanups, including STATE and SA_OPTIONS cleanup.

   * Key splitting now only across auth algorithm and encryption
     algorithm. Thanks for B. Sommerfeld for clues here.

   The following changes were made between 00 and 01:

   * Added this change log.

   * Simplified TLV header syntax.

   * Splitting of algorithms. This may be controversial, but it allows
     PF_KEY to be used for more than just IPsec. It also allows some
     kinds of policies to be placed in the KMd easier.

   * Added solid definitions and formats for certificate identities,
     multiple keys, etc.

   * Specified how keys are to be layed out (most-to-least bits).

   * Changed sequence number semantics to be like an RPC transaction ID
     number.

F. Full Copyright Statement