

PF_KEY Key Management API, Version 2

STATUS OF THIS MEMO

This document is an Internet Draft. Internet Drafts are working documents.

Internet Drafts are draft documents valid for a maximum of 6 months. Internet Drafts may be updated, replaced, or obsoleted by other documents at any time. It is not appropriate to use Internet Drafts as reference material or to cite them other than as "work in progress".

A future version of this draft will be submitted to the RFC Editor for publication as an Informational document.

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. an ISAKMP daemon or SKIP certificate discovery protocol daemon).

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"). The Key Engine and its structures incorporate the required security attributes for a session and are instances of the "Security Association" 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. [\[Sk191\]](#) This document describes Version 2 of PF_KEY. Version 1 was implemented in the first three 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.

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 believed that it was 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

In this document, the words that are used to define the significance of each particular requirement are usually capitalized. These words are:

- MUST

This word or the adjective "REQUIRED" means that the item is an absolute requirement of the specification.

- SHOULD

This word or the adjective "RECOMMENDED" means that there might exist valid reasons in particular circumstances to ignore this item, but the full implications should be understood and the case carefully weighed before taking a different course.

- MAY

This word or the adjective "OPTIONAL" means that this item is truly optional. One vendor might choose to include the item because a particular marketplace requires it or because it enhances the product, for example; another vendor may omit the same item.

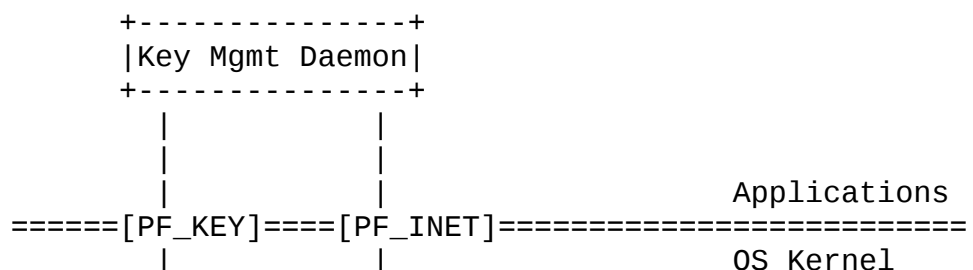
1.2 CONCEPTUAL MODEL

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 part at the application-layer. For example, the Photuris, ISAKMP, and Oakley proposals for IPsec key management are all application-layer protocols. 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, which it uses to communicate with the Key Engine, and PF_INET (or PF_INET6 in the case of IPv6), which it uses 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.



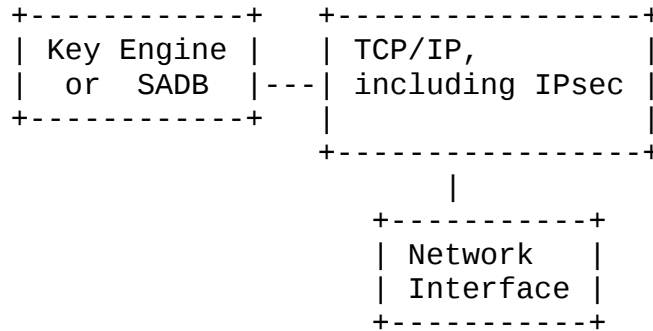


Figure 1: Relationship of Key Mgmt to PF_KEY

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.

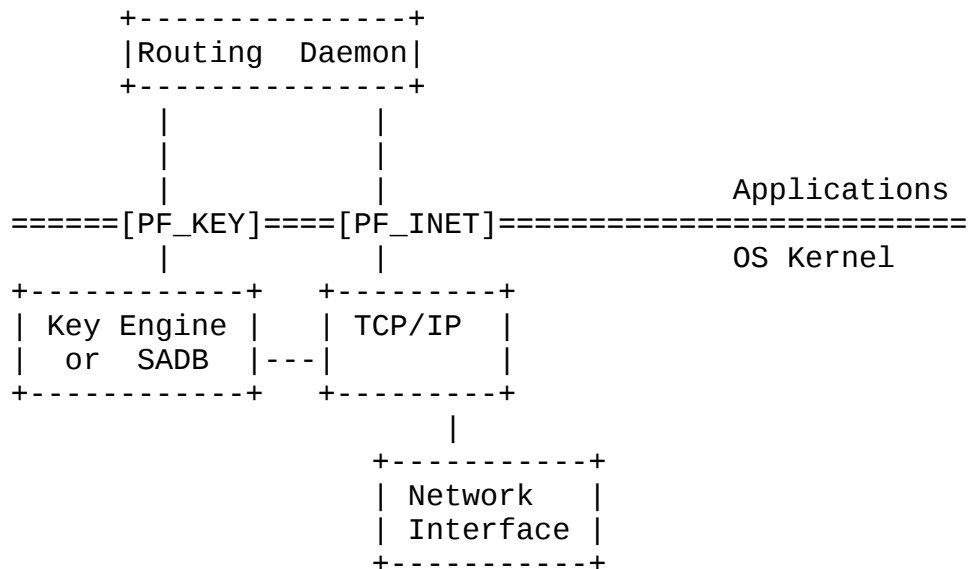


Figure 2: Relationship of Trusted Application to PF_KEY

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 a SADB_GET message to obtain the SA from the Key Engine.

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 socket is created as follows:

```
#include <netkey/key.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 the symbol PF_KEY_V2, which indicates to the kernel that the client would like to use Version 2 of the PF_KEY interface.

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 a privileged 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 messages sent to the kernel are returned to all open PF_KEY sockets, including the sender. A process can disable this looping back of messages it generates by disabling the SO_USELOOPBACK option using the setsockopt(2) call. A PF_KEY socket listener, which by default receives all replies may disable message reception by terminating socket input with the shutdown(2) call. 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 message from the kernel, in cases where not all PF_KEY sockets are in the same trust domain.

NOTE: SECTIONS LIKE THIS, INSIDE ***** ARE META-COMMENTS AND OPEN ISSUES THAT NEED CONTEXT TO BE CLEAR.

[RJA: Clarifying text on security restrictions is needed here, IMHO.]

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.

The second approach to add a new Security Association into the kernel is for the application to first request an SPI value from the kernel using the SADB_GETSPI message and then send a 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 a SADB_REGISTER message.

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 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 implementer debugging and is not used in ordinary operation of PF_KEY.

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 multi-octet fields. In the case that this becomes an issue, network byte order MUST be used.

2.1 BASE MESSAGE HEADER FORMAT

PF_KEY messages consist of the base message header followed by security association specific data whose lengths (and in some cases, types) are specified in the base message header itself. The data that follow the base header MUST follow in the order that their length fields appear in the security association base message header. A zero length implies that field is not present.

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

```

struct sadb_msg_hdr {
    u_int16_t      sadb_msg_len;
    u_int8_t       sadb_msg_type;
    u_int8_t       sadb_msg_errno;
    u_int32_t      sadb_msg_seq;
    u_int32_t      sadb_msg_pid;

    u_int8_t       sadb_sa_type;
    u_int8_t       sadb_sa_state;
    u_int8_t       sadb_sa_transform;

    u_int8_t       sadb_sa_srclen;
    u_int8_t       sadb_sa_dstlen;
    u_int8_t       sadb_sa_proxylen;
    u_int16_t      sadb_sa_keylen;
    u_int16_t      sadb_sa_ivlen;

    u_int8_t       sadb_proposal_len;
    u_int8_t       sadb_reserved;

    u_int32_t      sadb_sa_spi;
    u_int32_t      sadb_sa_sens_domain;
    u_int32_t      sadb_sa_typeopt;
    u_int32_t      sadb_sa_transopt;

    u_int8_t       sadb_sa_lifetype;
    u_int8_t       sadb_sa_lifetimelen;
    u_int8_t       sadb_sa_replay_window_len;
    u_int8_t       sadb_supported_xform_len;

    u_int8_t       sadb_sa_sens_label;
    u_int8_t       sadb_sa_sens_bitmap_len;
    u_int8_t       sadb_sa_integ_label;
    u_int8_t       sadb_sa_integ_bitmap_len;

    u_int8_t       sadb_sa_src_identtype;
    u_int8_t       sadb_sa_src_identlen;
    u_int8_t       sadb_sa_dst_identtype;
    u_int8_t       sadb_sa_dst_identlen;
};

```

`sadb_msg_seq` Contains a count of the requests originated by the sender. This field, along with `sadb_msg_pid`, can be used to uniquely identify requests to a process. The sender is responsible for filling in this field.

`sadb_msg_len` Contains the total length, in octets, of all data in the PF_KEY message including the 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 octets.

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 kernel stores the error code in this field if an error has occurred.

sadb_msg_pid Identifies the process which the kernel thinks the message is bound for. For example: If process id 2112 sends a SADB_UPDATE message to the kernel, the SADB_UPDATE reply from the kernel will fill in this field with 2112. This field, along with **sadb_msg_seq**, can be used to uniquely identify requests to a process.

It is currently believed that a 32-bit quantity will hold an operating system's process ID space. If this assumption is not true, then **sadb_msg_pid** will have to be revisited.

[Dan McD.: In BSD routing sockets, the `write()` or `sendto()` return `errno` immediately to the caller. In System V STREAMS, this is annoying. I propose that we say explicitly that applications MUST NOT depend on `errno` being returned immediately from `write()` and SHOULD use the **sadb_msg_errno** field instead.

There are also other properties of routing sockets (`pid`) that required the blotch we needed for the `errno`. I've consulted our routing sockets folks, and there are some issues. The PID issue isn't as harsh as the failing `write()` or `sendto()` calls, but both are issues in STREAMS. They are solvable, but annoying.]

sadb_sa_type indicates the type of security association (e.g. AH, ESP, OSPF, etc). Valid Security Association types are declared in the file `<netkey/key.h>`. The current set of Security Association types are enumerated later in this document.

sadb_sa_state Is a bitmask field containing the state of the Security Association. This field should be set to zero by the

sending process and is set to the state of the Security Association when the message is received. The current set of State flags are enumerated later in this document.

sadb_sa_transform

Identifies the cryptographic transform to use. The current set of transform names are enumerated later in this document. See [section 3.4](#) for values that can be placed in this field.

sadb_sa_srclen Contains the length of the source address for the security association. The source address is the address that will be present in the source address field of protocol headers to be processed by this security association. A value of zero indicates that no source address is present.

sadb_sa_dstlen Contains the length of the destination address for the security association. The destination address is the address that will be present in the destination address field of protocol headers to be processed by this security association. A value of zero indicates that no destination address is present.

sadb_sa_proxylen

Contains the length of the proxy address for the security association. The proxy address is the address of the remote system with which the security association was negotiated (i.e., the node that provided proxy security services on behalf of the system identified in the source address field). A length of zero indicates that proxy key management was not used for this SA and that no proxy address (i.e. "sadb_sa_proxy") will appear below.

sadb_sa_keylen Contains the length of the key for the security association, expressed in bits. The key is the data used by the cryptographic transform to process a packet, and it is usually secret. A value of zero indicates that no key is present.

sadb_sa_ivlen Contains the length of the initialization vector (IV) for the security association, expressed in bits. The IV is the random data used by the cryptographic transform to process a packet. The PF_KEY interface does not communicate or generate the IV data, but it does communicate the length of data needed. A value of zero indicates that

no IV is needed.

sadb_proposal_len

Size of the proposed_situation field. A size of zero indicates that field is not present in this message.

sadb_sa_spi

Contains the Security Parameters Index value for the Security Association. Although this is a 32-bit field, some types of Security Association 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.

sadb_sa_dpd_domain

Contains the "Data Protection Domain of Interpretation (DOI)" for the Sensitivity and Integrity data of this security association. A value of zero indicates that sensitivity/integrity labelling is not in use for this security association. This field contains a 32-bit unsigned integer.

Different organizations typically have different values for the Domain of Interpretation. A single organization can have multiple Domains of Interpretation (e.g. one for use by engineering and another for use by personnel) for the sensitivity labelling. See [Appendix A](#) for more on the DOI field.

The DOI value in a PF_KEY message is used with key management. In an environment where sensitivity labels are in use, communicating or negotiating the sensitivity/integrity information with the remote node can be an important part of the key management process. The Key Engine will typically need to map between the DOI value used with PF_KEY (and thence with key management) and the implementation-specific internal representation. See [Appendix A](#) for information on the semantics of DOI value

sadb_sa_typeopt Contains a bitmap of options (e.g. PFS) defined for each specific security association type.

sadb_sa_transopt

Contains a bitmap of options defined for the transform (e.g. replay protection) specified in the security association.

sadb_sa_lifetype

Contains the type of the lifetime information being

specified for this security association. A value of zero indicates that no lifetime information is being specified. Nonzero values indicate, for example, if the lifetime is in units of data, or in units of time.

sadb_sa_lifetimelen

Contains the length of each lifetime information datum being specified for this security association. A value of zero indicates that no lifetime information is being specified. If lifetime information is provided, two lifetimes MUST be specified: the first specifying a lower bound, or soft limit; and, the second specifying an upper bound, or hard limit. The hard limit must be greater than, or equal to the soft limit. The lifetime length must be large enough to hold the hard limit.

sadb_sa_replay_window_len

Size of the Replay Protection window in number of packets. Field is treated as unsigned. If replay protection is not in use, this will have value 0.

sadb_supported_xform_len

Size of the supported_transforms field. A size of zero indicates that field is not present in this message.

sadb_sa_sens_level

Indicates the hierarchical ("vertical") sensitivity level associated with this Security Association. The value 0 means "no associated sensitivity level". Values 1 through 255 are interpreted within the context of the sensitivity_domain of this Security Association. Higher values of sensitivity_level indicate greater sensitivity. For example "Top Secret" or "Mergers & Acquisitions" might have value 255 and "Unclassified but sensitive" or "employees only" might have value 10. This field is useful in enforcing any security policy following the Bell-LaPadula model.

sadb_sa_integ_level

Indicates the hierarchical ("vertical") integrity level associated with this Security Association. The value 0 means "no associated integrity level". Values 1 through 255 are interpreted within the context of the sensitivity_domain of this Security Association. Higher values of integrity level indicate greater integrity requirements. For example "Top Secret" or "Mergers & Acquisitions" might have value 255 and "Unclassified but sensitive" or "employees only" might

have value 10. This field is useful in supporting any integrity policy following the Biba model.

sadb_sa_sens_bitmap_len

Contains the length of the compartment bitmap of ("horizontal") sensitivity information being specified for this security association. A value of zero indicates that no sensitivity compartments are in use with this security association.

sadb_sa_integ_bitmap_len

Contains the length of the compartment bitmap of ("horizontal") integrity information being specified for this security association. A value of zero indicates that no integrity compartments are in use with this security association.

sadb_sa_src_identytype

sadb_sa_dst_identytype

Contains the type of the identity information being specified for this security association. A value of zero indicates that the source/destination addresses are the only identity information and that the corresponding identlen must also be zero. Source and/or destination identities may be specified.

sadb_sa_src_identlen

sadb_sa_dst_identlen

Contains the length of the identity certificate information being specified for this security association. A value of zero indicates that no identity information is being specified (other than the identity implicit in the source/destination addresses). Source and/or destination identities may be specified.

2.3 ADDITIONAL MESSAGE FIELDS

The additional data following the base header consists of the following variable length fields:

sadb_sa_src, sadb_sa_dst, sadb_sa_proxy

The src, dst, and proxy addresses are stored in socket address format. The header file <netkey/key.h> MUST define a symbol named KEY_SOCKADDR, which must resolve to the data type of the host's socket address structure. This socket address structure MUST either be compatible with 4.3BSD's struct sockaddr (no sa_len field) or with 4.4BSD's struct sockaddr (with an sa_len field). This will normally be defined as:

```
#define KEY_SOCKADDR    struct sockaddr
```

If the host OS provides the option of using 4.4BSD-style sockaddrs with an sa_len field, then they MUST be used with the PF_KEY interface. If the host OS does not normally define a socket address structure for use with its sockets interface, it MUST use the definition from 4.4BSD, which includes a sa_len field. The sa_len field in sockaddrs, while compatible with 4.4 BSD, is not strictly needed for PF_KEY sockaddrs, because the lengths are specified in the base header. A PF_KEY implementation MAY require its sockaddrs to have a valid value for sa_len.

All KEY_SOCKADDRs MUST have valid address family (sa_family) fields and, if appropriate, address length (sa_len) fields. All other fields in the sockaddr MUST be zeroed out if not used (e.g. sin_port, sin6_port, sin_zero, sin6_flowinfo, etc.). All additional data fields after the base header MUST be 32-bit word aligned to maintain overall message word alignment. Zero padding may be used. Padding MUST be ignored when processing messages. Zero length fields do not require padding.

When proxy key management was not used with a particular SA, then that SA will have no sadb_sa_proxy field in the PF_KEY messages or the proxy sockaddr will contain all zeros.

security_association_key

The cryptographic key to use with this security association. This MUST be padded to a 32-bit boundary.

[RJA: Triple-DES and other algorithms/transforms can require more than one

crypto key. How do we handle that case ?

Dan McD.: Also, if lengths are in octets, we can only support 2040 bits of key. If lengths are in 4-byte words, then we can support longer keys.]

life_time_soft, life_time_hard

These designators are in units specified by the sadb_sa_lifetype field and are each sadb_sa_lifelen octets in length. Allowable types of lifetimes are:

LIFETYPE_SEC	units of seconds of existence of the security association
LIFETYPE_KB	units of kilobytes of data to which the security association has been applied
LIFETIME_PACKETS	units of packets to which this SA has been applied. This might be used with Replay Protection.

sensitivity_compartment_bitmap

This is a variable-length array of octets, padded to a 32-bit boundary. The entire set of octets are treated as a single bitmap with each bit being associated with a particular ("horizontal") sensitivity compartment. The values must be interpreted in the context of the sensitivity_domain of this Security Association. For a given bit location, the value 0 means that compartment is not present in this Security Association while the value 1 means that compartment is present in this Security Association.

If sensitivity compartments are not in use, this field will not exist. Implementations on systems claiming to provide MLS MUST support compartment bitmaps of the larger of (A) at least 32 32-bit words or (B) the number of compartments natively supported on that MLS system. Systems not claiming to provide MLS SHOULD support system administrator configuration of a single static sens_compartment_bitmap value that can be used by key management applications communicating with MLS systems.

integrity_compartment_bitmap

This is a variable-length array of octets, padded to a 32-bit boundary. The entire set of octets are treated as a single bitmap with each bit being associated with a particular ("horizontal") integrity compartment. The values must be interpreted in the context of the sensitivity_domain of this Security Association. For a given bit location, the value 0

means that compartment is not present in this Security Association while the value 1 means that compartment is present in this Security Association. If integrity compartments are not in use for a session, this field will not exist. Systems SHOULD permit configuration of at least a single static integ_compartment_bitmap value that can be used by key management applications communicating with that system.

supported_transforms

This is a variable-length array of octets, padded to a 32-bit boundary. Each octet contains a single transform identifier. The value zero means "no transform" and is used only to pad to a 32-bit boundary. This field allows key management applications to know what transforms are supported by the kernel. This is used with the SADB_REGISTER message. The size of this field is indicated by the sadb_supported_xform_len field in the base header.

The transforms in this fields SHOULD be ordered in preferential order.

[RJA: Transform value assignment remains an open issue.

Dan McD: Let's also not forget that when Steve Kent rewhacks things, the representation of supported <blah> comes into question.

Also, I mention preferential order, but how do I order ESP and AH relative to each other? First the ESP xforms, then AH?]

src_identity, dst_identity

These fields denote identity certificate information for the parties using this security association. They are of the type specified by their corresponding identity type field and a length in octets specified by their corresponding identity length field.

Identity certificate information is supplied to the key management application for use in negotiation. Certain key management protocols (e.g. ISAKMP) allow for refined policy checks based upon the identities of the parties which will use the security association.

Allowable identity types are:

ID_IPV4_ADDR	an IPv4 address
ID_IPV4_ADDR_RANGE	an IPv4 address and netmask denoting a subnet
ID_IPV6_ADDR	an IPv6 address
ID_IPV6_ADDR_RANGE	an IPv6 address and netmask denoting a subnet
ID_FQDN	a fully-qualified domain name
ID_USER_FQDN	user@fully-qualified-domain-name
ID_IPV4_CONNID	A 5-tuple of <source IP addr, dest IP addr, source port, destination port, protocol number>. IP addr is IPv4 or IPv6 as appropriate. The protocol number is the same value as appears in the IP protocol field, or the IPv6 next-header field.
ID_IPV6_CONNID	

proposed_situation

This is a variable-length array of octets, padded to a 32-bit boundary. The term, "proposed situation," is borrowed from ISAKMP. A proposed situation is a list of possible algorithms and algorithm options (or transforms) which the kernel requests of key management.

The transforms in this field MUST be ordered in preferential order. This field only appears in outgoing SADB_ACQUIRE messages, and indicates that other transforms, in addition to the one in the base header, may be acceptable.

[Dan McD.: Same issues as supported_transforms, plus an open question about if we included lifetimes in here as well.]

2.4 ILLUSTRATION OF MESSAGE LAYOUT

The following shows how the octets are layed 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
      +-----+-----+-----+-----+
      |                                     sadb_msg_seq                                     |
      +-----+-----+-----+-----+
      |      sadb_msg_len      | sadb_msg_type | sadb_msg_errno |
      +-----+-----+-----+-----+
      |      sadb_msg_pid (NOTE: Assuming pid_t is 32 bits)      |
      +-----+-----+-----+-----+
      | sadb_sa_type | sadb_sa_state | ..transform | sadb_sa_srclen |
      +-----+-----+-----+-----+
      |sadb_sa_dstlen |sadb_sa_proxylen|sadb_sa_keylen| sadb_sa_ivlen |
      +-----+-----+-----+-----+
      |                                     sadb_sa_spi                                     |
      +-----+-----+-----+-----+
      |                                     sadb_sa_sens_domain                                     |
      +-----+-----+-----+-----+
      |                                     sadb_sa_typeopt                                     |
      +-----+-----+-----+-----+
      |                                     sadb_sa_transopt                                     |
      +-----+-----+-----+-----+
      | ..lifetype      | ..lifetimelen | ..sens_label | ..bitmap_len |
      +-----+-----+-----+-----+
      |..src_identtype| ..src_identlen|..dst_identtype| ..dst_identlen|
      +-----+-----+-----+-----+

```

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

```

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
+-----+-----+-----+-----+
<                                     <
|           Source sockaddr (optional)           |
>           (Padded to 32 bits)                   >
+-----+-----+-----+-----+

+-----+-----+-----+-----+
<                                     <
|           Destination sockaddr (optional)        |
>           (Padded to 32 bits)                   >
+-----+-----+-----+-----+

+-----+-----+-----+-----+
<                                     <
|           Proxy sockaddr (optional)              |
>           (Padded to 32 bits)                   >
+-----+-----+-----+-----+

+-----+-----+-----+-----+
<                                     <
|           Cryptographic Key (optional)           |
>           (Padded to 32 bits)                   >
+-----+-----+-----+-----+

+-----+-----+-----+-----+
<                                     <
|           Initialization vector (optional)       |
>           (Padded to 32 bits)                   >
+-----+-----+-----+-----+

+-----+-----+-----+-----+
<                                     <
|           Soft and hard lifetime                 |
>           (Padded to 32 bits)                   >
+-----+-----+-----+-----+

+-----+-----+-----+-----+
<                                     <
|           Compartment bitmap (optional)          |
>           (Padded to 32 bits)                   >
+-----+-----+-----+-----+

+-----+-----+-----+-----+
<                                     <
|           Supported Transforms (optional)        |
>           (Padded to 32 bits)                   >
+-----+-----+-----+-----+

```

```
+-----+-----+-----+-----+
<                                     <
|           Source Identity information (optional)           |
|           (Padded to 32 bits)                               |
>                                     >
+-----+-----+-----+-----+

+-----+-----+-----+-----+
<                                     <
|           Destination Identity information (optional)       |
|           (Padded to 32 bits)                               |
>                                     >
+-----+-----+-----+-----+
```

3. SYMBOLIC NAMES

This section defines various symbols used with PF_KEY and the semantics associated with each symbol. Applications should use the symbolic name in order to be maximally portable. The numeric definitions shown are for illustrative purposes, unless explicitly stated otherwise. The numeric definition might 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 <netkey/key.h>:

```
#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 by normal applications */
```

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) >

3.1.1 SADB_GETSPI Message

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 a 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 the source and destination sockaddrs, and, if proxy key management is in use, followed by the proxy sockaddr. The application also specifies the SPI. This is done by either setting the `sadb_sa_spi` field to a single SPI, or having the kernel select within a range of SPI values by setting the `sadb_sa_spi` value to 0, and setting the lowest value and highest value by using 4 octet lifetimes. Setting `sadb_sa_lifetype` to `LIFETYPE_SPI_RANGE` (seen earlier in this document along with other `LIFETYPE_*` values), setting the `sadb_sa_lifetimelen` to 4, and setting the 4-octet lifetime values to the lower and upper bounds for SPI selection. 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. An update message can later be used to add an entry with the requested SPI value.

The message behavior of the SADB_GETSPI message is:

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

<base, source, dest, (proxy,) (lifetimes) >

The kernel returns the SADB_GETSPI message to all listening processes.

<base, source, dest, (proxy) >

Errors:

- EINVAL Various message improprieties, including SPI ranges that are malformed.
- ENOBUFS No buffer space is available to process request.
- EEXIST Requested SPI or SPI range is not available/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 larval security association created with SADB_GETSPI. The format of the update message is a base header, followed by source, destination, and possibly proxy. If the key, IV, lifetimes, or compartment bitmap need to be updated, these fields should be included. 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 SHOULD perform sanity checking on various technical parameters passed in as part of the SADB_UPDATE message. One example is DES key parity bit checking. Other examples include key length checking, and checks for keys known to be weak for the specified algorithm.

The kernel SHOULD NOT allow 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 associations. Malicious trusted parties could still issue a 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 message behavior of the SADB_UPDATE message is:

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

<base, source, dest, (proxy,), key, lifetimes, (compartment bitmaps)

The kernel returns the SADB_UPDATE message to all listening processes.

<base, source, dest, (proxy,) >

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 Various message improprieties, including sanity check failures on keys.
- EACCES Insufficient privilege to update entry. Socket issuing

the SADB_UPDATE is not creator of the entry to be updated.

3.1.3 SADB_ADD Message

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 optimal for manual keying applications, and other strategies where the uniqueness of the SPI is known immediately.

The message behavior of the SADB_ADD message is:

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

<base, source, dest, (proxy,), key, lifetimes, (compartment bitmaps)

The kernel returns the SADB_ADD message to all listening processes.

<base, source, dest, (proxy,) (compartment bitmaps)>

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	Various message improprieties, including sanity check failures on keys.

3.1.4 SADB_DELETE Message

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 source sockaddr and the destination sockaddr. The kernel deletes the security association matching the type, spi, source address, and destination address in the message.

There are two message behaviors for SADB_DELETE. The first is a user- originated deletion

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

<base, source, dest>

The kernel returns the SADB_DELETE message to all listening processes.

<base, source, dest>

The second behavior is in the case of a hard-limit lifetime expiration.

The kernel sends a SADB_DELETE message to all listening processes when a security association times out.

<base, source, dest>

Errors:

ESRCH The security association to be deleted was not found.

3.1.5 SADB_GET Message

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 followed by the source sockaddr, destination sockaddr, and, if it is present in the requested SA, the proxy sockaddr. The Security Association matching the type, spi, source address, and destination address is returned. The K_USED flag is set inside the Key Engine for the returned Security Association.

The message behavior of the SADB_GET message is:

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

<base, source, dest, (proxy,)>

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

<base, source, dest, (proxy,) key, lifetimes,
(compartment bitmaps,) (source identity,) (dest identity)>

Errors:

ESRCH The sought security association was not found.

3.1.6 SADB_ACQUIRE Message

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 a source sockaddr, destination sockaddr, possibly certificate identity information, and if more than one algorithm and options is acceptable, a proposed situation. The source and destination sockaddr contains the source and destination address of the desired Security Association. The proposed situation contains a list of desirable algorithms and options that can be used if the algorithm and option in the base header is 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, except that the `sadb_pid`, `sadb_seq`, `sadb_errno`, and `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 by placing the most preferred algorithm in the base header, and the subsequent ones in the `proposed_situation` field in order of preference.

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

The kernel sends a SADB_ACQUIRE message to registered sockets.

```
<base, source, dest, (source certificate id,)
  (dest certificate id,) (proposed situation) >
```

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

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

```
<base, source, dest, (source certificate id,)
  (dest certificate id,) (proposed situation) >
```

The kernel returns a SADB_ACQUIRE message to registered sockets.

```
<base, source, dest, (source certificate id,)
  (dest certificate id,) (proposed situation) >
```

The user-level consumer waits for a 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 Message

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_sa_type`. 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. For portability reasons, key management applications MAY register for a type not known to the kernel.

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

The messaging behavior of the SADB_REGISTER message is:

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

<base>

The kernel returns a SADB_REGISTER message, with transform types supported by the kernel being indicated in the supported transforms field.

<base, supported_transforms>

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 of a Security Association has expired for a security protocol implemented inside the kernel, then the kernel MUST issue a SADB_EXPIRE message to all key socket listeners. A user application is responsible for tracking SA expirations for security protocols (e.g. OSPF Authentication) that are implemented inside that user application. If the soft limit of a Security Association has expired, the user application SHOULD issue a SADB_EXPIRE message. Regardless of where the security protocol is implemented, if both the soft limit and the hard limit expire at the same time, both SADB_DELETE and SADB_EXPIRE messages MUST be sent.

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

The messaging behavior of the SADB_EXPIRE message is:

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

<base, source, dest, (proxy,) (compartment bitmaps)>

ERRORS:

EINVAL Message Invalid for some reason.
EPROTONOSUPPORT ???

3.1.9 SADB_FLUSH Message

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

The messaging behavior for SADB_FLUSH is:

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

<base>

The kernel will return a 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 Message

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_sa_type` value is in the message, only associations of that type will be dumped. If `SEC_TYPE_NONE` is specified, all associations will be used. 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. Unlike other key messages, the dump message is returned only to the key socket originating the dump request because of the potentially large amount of data it can generate. 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 the key socket, hence key management applications MUST NOT depend on this message for basic operation.

The messaging behavior for SADB_DUMP is:

Send a 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, source, dest, (proxy,) (key,) (iv,) (lifetimes,) (source certificate id,) (dest certificate id) >

3.2 SECURITY ASSOCIATION STATE

The Security Association's state is a bitmask field. The related symbolic definitions below should be used in order that applications will be maximally portable:

```
#define SA_USED          0x01    /* SA used/not used */
#define SA_UNIQUE        0x02    /* SA unique/reusable */
#define SA_LARVAL        0x04    /* SPI assigned, but SA incomplete
#define SA_ZOMBIE        0x08    /* SA expired but still useable */
#define SA_DEAD          0x10    /* SA marked for deletion */
#define SA_INBOUND       0x20    /* SA for packets destined here */
#define SA_OUTBOUND      0x40    /* SA for packets sourced here */
#define SA_FORWARD       0x80    /* SA for packets forwarded thru */
```

All unspecified values in the bitmask field are reserved and MUST NOT be used.

SA_USED is set by the operating system if the Security Association has been used. Otherwise this flag is not set. If SADB_GET is used to read an SA from the Key Engine, the Key Engine will set SA_USED on the SA that was read via SADB_GET.

SA_UNIQUE is set by the operating system if the Security Association has been allocated uniquely to a single user (e.g. a particular network socket). If this is not set, then the Security Association is considered sharable.

SA_LARVAL indicates that the operating system has assigned this SPI value but that there is no complete Security Association yet stored in the kernel.

SA_ZOMBIE indicates a Security Association that has expired but is still useable until a replacement Security Association is added. This is primarily used with OSPFv2 and RIPv2 cryptographic authentication.

SA_DEAD indicates a Security Association that exists but is marked for deletion.

SA_INBOUND is set for an inbound Security Association and SA_OUTBOUND is set for an outbound Security Association. SA_FORWARD is used for a Security Association used only for packets originating elsewhere and destined elsewhere that have security processing on this node. All Security Associations used with PF_KEY are unidirectional.

***** [Dan McD.: Why SA_FORWARD? Isn't SA_OUTBOUND
sufficient? I may implement forwarding such that

it's hard or impossible for
tell the difference.] *****

the key engine/SADB to

3.3 SECURITY ASSOCIATION TYPE

This defines the type of Security Association in this message. The numeric definitions are those used in the prototype NRL implementation, but might be different on other implementations. 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 <netkey/key.h>.

```
#define SEC_TYPE_NONE          0

#define SEC_TYPE_AH            1 /* RFC-1826 */
#define SEC_TYPE_ESP          2 /* RFC-1827 */

#define SEC_TYPE_RSVP          3 /* RSVP Authentication */
#define SEC_TYPE_OSPFV2        4 /* OSPFv2 Authentication */
#define SEC_TYPE_RIPV2         5 /* RIPv2 Authentication */
#define SEC_TYPE_MIPV4         6 /* Mobile IPv4 Authentication */

#define SEC_TYPE_MAX           6
```

SEC_TYPE_NONE is defined for completeness and means no Security Association. This type is never used with PF_KEY.

SEC_TYPE_AH is for the IP Authentication Header defined in [[Atk95b](#)]. SEC_TYPE_ESP is for the IP Encapsulating Security Payload defined in [[Atk95c](#)].

SEC_TYPE_RSVP is for the RSVP Integrity Object.

SEC_TYPE_OSPFv2 is for OSPFv2 Cryptographic authentication, while SEC_TYPE_RIPv2 is for RIPv2 Cryptographic authentication.

SEC_TYPE_MAX is never used with PF_KEY but is defined for completeness. It is always set to the highest valid numeric value. There must not be gaps in the numbering of security types; all numbers must be used sequentially.

3.4 ALGORITHM TYPE

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 editor.

The symbols below are defined in <netkey/key.h>.

```
#define SEC_ALGTYPE_AH_MD5          0 /* deprecated */
#define SEC_ALGTYPE_AH_SHA          1 /* deprecated */
#define SEC_ALGTYPE_AH_MD5_HMAC     2
#define SEC_ALGTYPE_AH_SHA1_HMAC    3

#define SEC_ALGTYPE_ESP_DES_CBC     0 /* deprecated */
#define SEC_ALGTYPE_ESP_3DES        1 /* deprecated */
#define SEC_ALGTYPE_ESP_COM_DES_CBC_MD5 2

#define SEC_ALGTYPE_RSVP_MD5        0
#define SEC_ALGTYPE_OSPF_MD5        0
#define SEC_ALGTYPE_RIP_MD5         0
```

[RJA: With the coming Steve Kent editorial changes to IPsec, this section is likely to change slightly to just specify "algorithms" and not specify "transforms".

Dan McD: Given this, the "transform options" become MUCH more critical.]

The algorithm for SEC_ALGTYPE_AH_MD5_HMAC is defined in [[OG96](#)]. The algorithm for SEC_ALGTYPE_AH_SHA1_HMAC is defined in [[CG96](#)]. The algorithm for SEC_ALGTYPE_ESP_COM_DES_CBC_MD5 is defined in [[Hug96](#)].

Mobile IP does not yet have a named algorithm type.

3.5 TYPE OPTIONS

Security association types can have various options defined. Options are denoted by a bit setting in the "Type Options" field of

the base header. The bitmasks for defined options MUST NOT vary between implementations.

```
#define SEC_TYPEOPTION_PFS    0x00000001 /* Use Perfect Forward Secrecy */
```

The SEC_TYPEOPTION_PFS flag indicates to key management that this association should have perfect forward secrecy in its key. (In other words, the session key cannot be determined by cryptanalysis of previous keying material.)

3.6 TRANSFORM OPTIONS

Algorithm-specific options are specified by the "Transform Options" bitmap of the base header. The bitmasks for defined options MUST NOT vary between applications. Note that if SEC_TRANSOPTION_TUNNEL is not set for an IPsec SA, then that SA must be a transport-mode SA.

```
#define SEC_TRANSOPTION_REPLAY    0x00000001 /* Replay Protection enabled
#define SEC_TRANSOPTION_TUNNEL    0x00000002 /* Tunnel Mode enabled */
#define SEC_TRANSOPTION_AH_PAD    0x00000004 /* Pad AH to 64-bit boundary
```

```
***** [Dan McD.:      We may need another one SEC_TRANSOPTION_HMAC,
for ESP, and           if that's the case, we may need a
secondary algorithm    identifier for ESP. ] *****
```

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.

```
*****
[RJA:  What else belongs here ? ]
*****
```

5. SECURITY CONSIDERATIONS

This draft discusses a method for creating, reading, 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.

ACKNOWLEDGEMENTS

The editors of this document are listed primarily in Alphabetical order. A side effect of this particular alphabetical listing is to also show the history (starting with the most recent) of text contribution to this document. Ran Atkinson also contributed much advice and wisdom toward this document.

REFERENCES

- [AMPMC96] Randall J. Atkinson, Daniel L. McDonald, Bao G. Phan, Craig W. Metzger, 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] Randall J. Atkinson, IP Security Architecture, [RFC-1825](#), August 1995.
- [Atk95b] Randall J. Atkinson, IP Authentication Header, [RFC-1826](#), August 1995.
- [Atk95c] Randall J. Atkinson, IP Encapsulating Security Payload, [RFC-1827](#), August 1995.
- [CG96] S. Chang & Rob Glenn, "HMAC-SHA IP Authentication with Replay Prevention", Internet Draft, May 1996.
- [DIA] US Defense Intelligence Agency (DIA), "Compartmented Mode Workstation Specification", Technical Report DDS-2600-6243-87.
- [Hug96] Jim Hughes (Editor), "Combined DES-CBC, HMAC, and Replay Prevention Security Transform", Internet Draft, April 1996.
- [OG96] Mike Oehler & Rob Glenn, "HMAC-MD5 IP Authentication with Replay Prevention", Internet Draft, May 1996.
- [Sk191] 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 judgement 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.

EDITOR INFORMATION

Daniel L. McDonald
Sun Microsystems, Inc.
2550 Garcia Avenue, MS UMPK17-202
Mountain View, CA 94043-1100
E-mail: danmcd@eng.sun.com

Craig W. Metz
The Inner Net
Code 1123, Box 10314
Blacksburg, VA 24062-0314
E-mail: cmetz@inner.net

Bao G. Phan
U. S. Naval Research Laboratory
Code 5544
4555 Overlook Ave. SW
Washington, DC 20375
E-mail: phan@itd.nrl.navy.mil

APPENDIX A: DATA PROTECTION DOI ASSIGNMENTS

The Data Protection Domain of Interpretation (DP/DOI) indicates the context in which particular values of a Sensitivity Label, Integrity Label, Sensitivity Bitmap, and Integrity Bitmap are interpreted. This is a 32-bit opaque value.

If the highest order bit of the DP/DOI is set to 1, then the DP/DOI is not necessarily globally unique and is from a number space set aside for private use among consenting users.

If the highest order bit of the DP/DOI is set to zero, the DP/DOI is globally unique from a number space administered by the Internet Assigned Numbers Authority. In order to conserve the limited amount of globally unique DP/DOI number space, IANA will not normally permit any one organization to obtain very many DP/DOI values. The all zeros DP/DOI value is permanently reserved to mean that "no DP/DOI is in use".

INITIAL VALUES:

Organization:	Value:
-----	-----
No DP/DOI is in use	0
US Dept of Defense GENSER	1
US Dept of Defense SCI	2
US Dept of Energy	3
NATO	4

(REMAINDER OF THIS SECTION IS TBD)