U. Schilcher
H. Tschofenig
F. Muenz
Siemens AG

July 17, 2005

MOBIKE Extensions for PF_KEY
draft-schilcher-mobike-pfkey-extension-01.txt

Status of this Memo

Copyright Notice

Abstract

PF_KEY is a generic key management API used for communication between a trusted user level key management daemon and a key engine within the operating system. With the extension of IKEv2 for mobility and multihoming (MOBIKE) the existing capabilities of PF_KEY with regard to the creation, maintenance and deletion of security associations became insufficient. This document defines an extension to update an entity in the security association database. Additionally, it is

necessary to reflect the newly offered integrity and encryption algorithms with IKEv2 in PF_KEY.

Table of Contents

1. Introduction

PF_KEY [1] is a generic key management API used for communication between a trusted user level key management daemon and a key engine within the operating system. With the extension of IKEv2 for mobility and multihoming (MOBIKE) [12] the existing capabilities of PF_KEY with regard to the creation, maintenance and deletion of security associations became insufficient. If an IKEv2 implementation [13] supports MOBIKE, some additional interaction with the SAD and the SPD has to be provided. This includes additional operations on the security policy database (SPD), such as creation, update and deletion of SPD entries, and the possibility to update addresses for already existing SAs in the security association database (SAD). Since the PF_KEY interface in the current version does not support this operations, some extensions have to be defined.

This document is partially based on PF_KEY extensions provided the KAME stack (see also [14]), which go beyond those described in [1]. The authors think that it is necessary to update the original RFC 2367 PF_KEY version to reflect the state-of-the-art implementations.

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [2].

3. IPsec SA Update

The first extension allows an IKEv2 implementation to update the addresses of an existing security association (SA) dynamically. Updating IPsec SAs is one of the side-effect of the IKE-SA update, a feature provided by MOBIKE [12]. PF_KEY defines a number of messages, namely SADB_GETSPI, SADB_UPDATE, SADB_ADD, SADB_DELETE, SADB_GET, SADB_ACQUIRE, SADB_REGISTER, SADB_EXPIRE, SADB_FLUSH and SADB_DUMP, for interaction between the key management daemon and the key engine in the operating system.

In Section 3.1.2 of [1] an SADB_UPDATE message is described for updating all data stored for an existing SA. The only parameters, which cannot be updated using an SADB_UPDATE message, are the Security Parameter Index (SPI), the source and destination IP addresses. The reason for this design decision might be based on the IPsec SA identification, which included these parameters to uniquely select a given security association. This aspect can, however, be seen as historic. In IKEv2, without the use of MOBIKE, theses parameters would not change.

To allow an IKEv2 key management daemon to change the addresses of an existing SA, a new message type has to be introduced: SADB_X_ADDRUPDATE. The notation of SADB_X is intended to outline an extention to the current API defined in [1]. Required symbols or structures in the PF_KEYv2 name space that are not described in [1] should therefore start with "SADB_X_" or "sadb_x_".

The format of the SADB_X_ADDRUPDATE message is:

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

The kernel responds with a message of the form:

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

The meaning of the payloads of the message is the following: "base" defines the default message header, "SA(*)" identifies the security association to be updated, where (*) indicates that the SA payload contains only the SPI of it, "address(SD)" contains the source and the destination addresses of the existing SA and "new_address(SD)" the new source and destination addresses. For a more detailed description of the payloads see [1]. For the new_address(SD) attribute new payload types SADB_X_EXT_NEW_ADDRESS_SRC and SADB_X_EXT_NEW_ADDRESS_DST are needed. These payloads have the same content as the SADB_EXT_ADDRESS_SRC and SADB_EXT_ADDRESS_DST payloads.

If the kernel receives a SADB_X_ADDRUPDATE message it immediately updates the SA identified by the SPI in the message. If more than one SA has to be updated, several SADB_X_ADDRUPDATE messages have to be sent since each SA payload can only contain one SPI.

In an error case, like for instance a malformed message, the kernel will respond with:

<base>

The "errno" field of the message will provide further information about the error.

4. SA Extension

In case a protected packet arrives with an unknown SPI value, for which no corresponding SA exists, the kernel actively sends a SADB_ACQUIRE to all listening applications. Using the information given in the SADB_ACQUIRE, the applications are able to quickly create a SA, while the triggering packet is still in the kernel buffer. The important information that are missing, are the traffic selector (TS) addresses, which are negotiated by IKEv2 using the TS payloads.

Since the TS addresses are only stored inside the SPD, they have to be read from there (see section Section 5). For that purpose the ID, which identifies the SPD entry, to which the new SA corresponds, has to be known. The proposed way to pass that ID from the kernel to the IKEv2 implementation is in using the following extension of the PF_KEY interface.

An SA2 payload has to be included in the SADB_ACQUIRE message, which has to following content:

```
struct sadb_x_sa2 {
        uint16_t        sadb_x_sa2_len;
        uint16_t        sadb_x_sa2_exttype;
        uint8_t         sadb_x_sa2_mode;
        uint8_t         sadb_x_sa2_reserved1;
        uint16_t        sadb_x_sa2_reserved2;
        uint32_t        sadb_x_sa2_sequence;
        uint32_t        sadb_x_sa2_reqid;
} __attribute__((packed));
/* sizeof(struct sadb_x_sa2) == 16 */
```

sadb_x_sa2_len:

The sadb_x_sa2_len contains the length of the structure in 8 Byte blocks.

sadb_x_sa2_exttype:

This field contains the value identifying the SADB_X_SA2 payload.

sadb_x_sa2_mode:

The sadb_x_sa2_mode field identifies the IPsec mode (i.e., tunnel or transport mode).

sadb_x_sa2_sequence:

The sadb_x_sa2_sequence field contains the ID of the corresponding SPD entry.

sadb_x_sa2_reqid:

The request ID for that message.

This payload can also be added to SADB_ADD and SADB_UPDATE messages to tell the kernel whether the SA to be generated is a transport or a tunnel mode SA. If no SADB_X_SA2 payload is present, all SAs created will only support tunnel mode.

5. SPD Update

For manipulating SPD entries, new PF_KEY messages have to be introduced (see also the KAME IPsec implementation).

Note that specifying SPD updates is problematic since the KAME IPsec extensions have never been standardized. As a consequence, this text does not extend PF_KEY [1] itself.

These message types are quite similar to the message types used to manipulate the entries in the SAD. The following new message types are needed:

SADB_X_SPDADD:

To add a new entry to the SPD, the key management daemon needs to send a SADB_X_SPDADD message to the kernel. The format of the message is:

<base, policy, address(SD), [lifetime(HS)]>

The kernel responds with a message of the form:

<base, policy, address(SD), [lifetime(HSC)]>

The meaning of the payloads, except for the policy payload, can be found in [1]. The policy payload contains all specific information about the new entry:

```
struct sadb_x_policy {
        uint16_t        sadb_x_policy_len;
        uint16_t        sadb_x_policy_exttype;
        uint16_t        sadb_x_policy_type;
        uint8_t         sadb_x_policy_dir;
        uint8_t         sadb_x_policy_reserved;
        uint32_t        sadb_x_policy_id;
        uint32_t        sadb_x_policy_reserved2;
} __attribute__((packed));
/* sizeof(struct sadb_x_policy) == 16 */
```

The sadb_x_policy_len field contains the length of the payload in 8 Byte blocks and sadb_x_policy_exttype contains the value SADB_X_SPDADD. The type of the SA is indicated by the sadb_x_policy_type field (e.g., IPsec SA) and the sadb_x_policy_dir field indicates the direction of the SA (the possibilities are IPSEC_DIR_INBOUND, IPSEC_DIR_OUTBOUND and IPSEC_DIR_FWD). The sadb_x_policy_id field contains a value which

is unique for each SPD entry. It should be set to zero for a SADB_X_SPDADD message, since the kernel is going to fill this value in. This structure is followed by one or more ipsecrequest structures, one for each protocol used by the new SPD entry:

```
struct sadb_x_ipsecrequest {
        uint16_t        sadb_x_ipsecrequest_len;
        uint16_t        sadb_x_ipsecrequest_proto;
        uint8_t         sadb_x_ipsecrequest_mode;
        uint8_t         sadb_x_ipsecrequest_level;
        uint16_t        sadb_x_ipsecrequest_reserved1;
        uint32_t        sadb_x_ipsecrequest_reqid;
        uint32_t        sadb_x_ipsecrequest_reserved2;
} __attribute__((packed));
/* sizeof(struct sadb_x_ipsecrequest) == 16 */
```

sadb_x_ipsecrequest_len:

The sadb_x_ipsecrequest_len again contains the length of the structure including optional extensions, but this time in bytes.

sadb_x_ipsecrequest_proto:

The sadb_x_ipsecrequest_proto field identifies the protocol used for the current structure (e.g., ESP or AH).

sadb_x_ipsecrequest_mode:

The sadb_x_ipsecrequest_mode field identifies the IPsec mode (i.e., tunnel or transport mode), which can be different for each protocol.

sadb_x_ipsecrequest_level:

The sadb_x_ipsecrequest_level field contains one of the following values: 'default', 'use', 'require' or 'unique'. It defines how and when a corresponding SA is used. The value 'use' means that an SA is used if available, otherwise the kernel keeps its normal operation. If 'require' is specified, it means that an SA is required for each packet matching to the policy entry. The value 'unique' has the same meaning as require except that the policy entry is bound to exactly one outbound SA.

sadb_x_ipsecrequest_reqid:

An ID for that SA can be passed to the kernel in the sadb_x_ipsecrequest_reqid field.

If tunnel mode is specified, the sadb_x_ipsecrequest structure is followed by two sockaddr structures that define the tunnel endpoint addresses. In the case that transport mode is used, no additional addresses are specified. The next payloads of the message are the source and destination addresses of the communication to be protected. In tunnel mode it is possible to use address ranges instead of single address pairs to protect the traffic of whole subnets with one SPD entry. It is also possible to specify hard and soft lifetimes for policy entries, but these payloads are optional. In the response from the kernel a hard, a soft and a current lifetime are always present. The semantics are the same as for SAD entries (see [1]).

SADB_X_SPDUPDATE:

If an existing SPD entry should be updated, the IKEv2 implementation sends a SADB_X_SPDUPDATE message to the kernel. This massage has the following format:

<base, policy, address(SD),[lifetime(HS)]>

The kernel responds with a message of the form:

<base, policy, address(SD), [lifetime(HSC)]>

The meaning of the payloads is the same as for the SADB_X_SPDADD message. All the content of a SPD entry can be changed except the sadb_x_policy_id field and the source/destination addresses, which are the inner addresses in tunnel mode. However, the tunnel endpoint addresses, which only exist in tunnel mode, can be changed using a SADB_X_SPDUPDATE message.

SADB_X_SPDDELETE:

A SADB_X_SPDDELETE message is sent to the kernel in the case that an existing SPD entry should be deleted. The entry is identified by the policy data and the source and destination address. The message has the following format:

<base, policy, address(SD)>

The kernel responds with a message of the form:

<base, policy, address(SD), [lifetime(HSC)]>

If no corresponding entry can be found, the kernel returns a message containing only the base header with the errno value set appropriately.

SADB_X_SPDGET:

If the content of an existing SPD entry is needed, a SADB_X_SPDGET message has to be sent to the kernel. The entry is identified by the sadb_x_policy_id entry in the sadb_x_policy structure. This id can obtained for example from a SADB_ACQUIRE message. The format of a SADB_X_SPDGET message is:

<base, policy>

The kernel responds with a message of the form:

<base, policy, address(SD), [lifetime(HSC)]>

If no entry has been found, the kernel returns an errno value in the base header.

SADB_X_SPDDUMP:

If the kernel receives a SADB_X_SPDDUMP message, it prints out all existing SPD entries on the console. The message format is:

<base>

SADB_X_SPDFLUSH:

To delete all SPD entries a SADB_X_SPDFLUSH message has to be sent to the kernel. The format of the message is:

<base>

6. Algorithm Types

This document defines an IANA registry for the IKEv2 defined cryptographic algorithms and thereby extends the algorithms defined by PF_KEY (see Section 3.5 of [1]). The same set of algorithms is available to MOBIKE.

The following algorithms have been defined already in PF_KEY, Section 3.5 of [1]):

/* Integrity (Authentication) Algorithms */

PF_KEY Algorithm Name	Value	Description
SADB_AALG_NONE	0	not used
SADB_AALG_MD5HMAC	2	HMAC-MD5-96
SADB_AALG_SHA1HMAC	3	HMAC-SHA-1-96

/* Encryption Algorithms */

PF_KEY Algorithm Name	Value	Description
SADB_EALG_NONE	0	not used
SADB_EALG_DESCBC	2	DES in CBC mode
SADB_EALG_3DESCBC	3	TripleDES in CBC mode
SADB_EALG_NULL	11	NULL encryption

The algorithm for SADB_AALG_MD5_HMAC is defined in [3]. The algorithm for SADB_AALG_SHA1HMAC is defined in [4]. The algorithm for SADB_EALG_DESCBC is defined in [5]. SADB_EALG_NULL is the NULL encryption algorithm, defined in [6]. 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).

This document enhances this list with the following algorithms:

/* Integrity (Authentication) Algorithms */

PF_KEY Algorithm Name	Value	Description
SADB_AALG_AESXCBCMAC	4	AES-XCBC-MAC-96
SADB_X_AALG_SHA2_256HMAC	5	SHA2-HMAC-256
SADB_X_AALG_SHA2_384HMAC	6	SHA2-HMAC-384
SADB_X_AALG_SHA2_512HMAC	7	SHA2-HMAC-512
SADB_X_AALG_RIPEMD160HMAC	8	HMAC-RIPEMD-160-96

/* Encryption algorithms */

PF_KEY Algorithm Name	Value	Description
SADB_EALG_AESCBC128	12	AES with 128-bit keys in CBC mode
SADB_X_EALG_CASTCBC	6	CAST in CBC mode
SADB_X_EALG_BLOWFISHCBC	7	BLOWFISH in CBC mode
SADB_X_EALG_AESCBC	12	AES in CBC mode
SADB_X_EALG_AESCTR	13	AES Counter Mode

AES-XCBC-MAC-96 is defined in [7] and AES with 128-bit keys in CBC mode is defined in [8]. AES counter mode has been defined for usage with IPsec ESP (see [9]). HMAC-RIPEMD-160-96 is defined in [10].

Note that compression algorithms also need to be considered. This document does not list them, however.

7. Traffic Selector Extensions

Information about Traffic Selectors should also be added to a updated version of PF_KEY [1]. This is left for future work.

8. IANA Considerations

This document defines an IANA registry for the cryptographic algorithms used within PF_KEY:

TBD

9. Security Considerations

This document describes an extension to PF_KEY [1] and therefore inherits its security properties. Since this interface allows existing entries in the security association database (and the security policy database) to be created, updated or deleted it needs to be ensured that only trusted and privileged processes are allowed to this interface.

10. Acknowledgments

The authors would like to thank Bao G. Phan for his initial PF_KEY implementation at US Naval Research Lab and the developers of FreeBSD for providing their PF_KEY implementation and for extending it for policy support, as well as R.J. Atkinson and Dan McDonald.

11. References

11.1 Normative References

[1] McDonald, D., Metz, C., and B. Phan, "PF_KEY Key Management API, Version 2", RFC 2367, July 1998.

[2] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", March 1997.

[3] Madson, C. and R. Glenn, "The Use of HMAC-MD5-96 within ESP and AH", RFC 2403, November 1998.

[4] Madson, C. and R. Glenn, "The Use of HMAC-SHA-1-96 within ESP and AH", RFC 2404, November 1998.

[5] Madson, C. and N. Doraswamy, "The ESP DES-CBC Cipher Algorithm With Explicit IV", RFC 2405, November 1998.

[6] Glenn, R. and S. Kent, "The NULL Encryption Algorithm and Its Use With IPsec", RFC 2410, November 1998.

[7] Frankel, S. and H. Herbert, "The AES-XCBC-MAC-96 Algorithm and Its Use With IPsec", RFC 3566, September 2003.

[8] Frankel, S., Glenn, R., and S. Kelly, "The AES-CBC Cipher Algorithm and Its Use with IPsec", RFC 3602, September 2003.

[9] Housley, R., "Using Advanced Encryption Standard (AES) Counter Mode With IPsec Encapsulating Security Payload (ESP)", RFC 3686, January 2004.

[10] Keromytis, A. and N. Provos, "The Use of HMAC-RIPEMD-160-96 within ESP and AH", RFC 2857, June 2000.

[11] Hoffman, P., "Cryptographic Suites for IPsec", draft-ietf-ipsec-ui-suites-06 (work in progress), April 2004.

11.2 Informative References

[12] Kivinen, T. and H. Tschofenig, "Design of the MOBIKE protocol", draft-ietf-mobike-design-02 (work in progress), February 2005.

[13] Kaufman, C., "Internet Key Exchange (IKEv2) Protocol", draft-ietf-ipsec-ikev2-17 (work in progress), October 2004.

[14] , ., "PF_KEY Extensions for IPsec Policy Management in KAME Stack, available at http://www.kame.net/newsletter/20021210/

(February 2005)", 12 2002.

Authors' Addresses

Udo Schilcher
Siemens AG
Otto-Hahn-Ring 6
Munich, Bayern 81739
Germany

Email: udo.schilcher@edu.uni-klu.ac.at

Hannes Tschofenig
Siemens AG
Otto-Hahn-Ring 6
Munich, Bayern 81739
Germany

Email: Hannes.Tschofenig@siemens.com

Franz Muenz
Siemens AG
Otto-Hahn-Ring 6
Munich, Bayern 81739
Germany

Email: Franz.Muenz@thirdwave.de

Intellectual Property Statement

Disclaimer of Validity

Copyright Statement

Acknowledgment

Funding for the RFC Editor function is currently provided by the Internet Society.