DECLARATION OF A. JEAN MAHONEY FOR IETF ADMINISTRATION LLC (IETF)

RFC 4251: The Secure Shell (SSH) Protocol Architecture;
RFC 4252: The Secure Shell (SSH) Authentication Protocol;
RFC 4253: The Secure Shell (SSH) Transport Layer Protocol;
RFC 4301: Security Architecture for the Internet Protocol;
RFC 4302: IP Authentication Header;
RFC 6101: The Secure Sockets Layer (SSL) Protocol Version 3.0

I, A. Jean Mahoney, hereby declare that all statements made herein are of my own knowledge and are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code:

1. I am an employee of Association Management Solutions, LLC (AMS), which acts under contract to the IETF Administration LLC (IETF) as the operator of the RFC Production Center. The RFC Production Center is part of the "RFC Editor" function, which prepares documents for publication and places files in an online repository for the authoritative Request for Comments (RFC) series of documents (RFC Series), and preserves records relating to these documents. The RFC Series includes, among other things, the series of Internet standards developed by the IETF. I hold the position of Associate Director of the RFC Production Center. I began employment with AMS on 26 August 2019.

2. My responsibilities as Associate Director of the RFC Production Center include acting as the custodian of records relating to the RFC Series, and I am familiar with the record keeping practices relating to the RFC Series, including the creation and maintenance of such records.

3. I have held my position as Associate Director with the RFC Production Center since 1 January 2023. Prior to becoming the Associate Director, I held various manager and editor positions with AMS.

4. The RFC Editor function was conducted by the Information Sciences Institute at the University of California ("ISI") under contract to the United States government prior to 1998. In 1998, the Internet Society (ISOC), in furtherance of its IETF activity, entered into the first in a series of contracts with ISI providing for ISI's performance of the RFC Editor function. Beginning in 2010, certain aspects of the RFC Editor function were assumed by the RFC Production Center operation of AMS under contract to ISOC (acting through its IETF function and, in particular, the IETF Administrative Oversight Committee (now the IETF Administration LLC). The business records of the RFC Editor function, as it was conducted by ISI, are currently housed with a cloud vendor under contract with IETF Administration LLC.

5. I make this declaration based on my personal knowledge and information contained in the business records of the RFC Editor as they are currently housed by AMS with a cloud vendor under contract with IETF Administration LLC, or in confirmation with other responsible RFC Editor personnel with such knowledge.

6. Prior to 1998, the RFC Editor's regular practice was to publish RFCs, making them available from a repository via FTP. When a new RFC was published, an announcement of its publication, with information on how to access the RFC, would be typically sent out within 24 hours of the publication.

7. Since 1998, the RFC Editor's regular practice was to publish RFCs, making them available on the RFC Editor website or via FTP. When a new RFC was published,

an announcement of its publication, with information on how to access the RFC, would be typically sent out within 24 hours of the publication. The announcement would go out to all subscribers and a contemporaneous electronic record of the announcement is kept in the IETF mail archive that is available online.

8. Beginning in 1998, any RFC published on the RFC Editor website or via FTP was reasonably accessible to the public and was disseminated or otherwise available to the extent that persons interested and ordinarily skilled in the subject matter or art exercising reasonable diligence could have located it. In particular, the RFCs were indexed and placed in a public repository.

9. The RFCs are kept in an online repository in the course of the RFC Editor's regularly conducted activity and ordinary course of business. The records are made pursuant to established procedures and are relied upon by the RFC Editor in the performance of its functions.

10. It is the regular practice of the RFC Editor to make and keep the RFC records.

11. Based on the business records for the RFC Editor and the RFC Editor's course of conduct in publishing RFCs, I have determined that the publication date of RFC 4251 (The Secure Shell (SSH) Protocol Architecture) was no later than January, 2006, at which time it was reasonably accessible to the public either on the RFC Editor website or via FTP from a repository. An announcement of its publication also would have been sent out to subscribers within 24 hours of its publication. A copy of that RFC is attached to this declaration as **Exhibit 1**.

12. Based on the business records for the RFC Editor and the RFC Editor's

course of conduct in publishing RFCs, I have determined that the publication date of RFC 4252 (The Secure Shell (SSH) Authentication Protocol) was no later than January, 2006, at which time it was reasonably accessible to the public either on the RFC Editor website or via FTP from a repository. An announcement of its publication also would have been sent out to subscribers within 24 hours of its publication. A copy of that RFC is attached to this declaration as **Exhibit 2**.

13. Based on the business records for the RFC Editor and the RFC Editor's course of conduct in publishing RFCs, I have determined that the publication date of RFC 4253 (The Secure Shell (SSH) Transport Layer Protocol) was no later than January, 2006, at which time it was reasonably accessible to the public either on the RFC Editor website or via FTP from a repository. An announcement of its publication also would have been sent out to subscribers within 24 hours of its publication. A copy of that RFC is attached to this declaration as **Exhibit 3.**

14. Based on the business records for the RFC Editor and the RFC Editor's course of conduct in publishing RFCs, I have determined that the publication date of RFC 4301 (Security Architecture for the Internet Protocol) was no later than December, 2005, at which time it was reasonably accessible to the public either on the RFC Editor website or via FTP from a repository. An announcement of its publication also would have been sent out to subscribers within 24 hours of its publication. A copy of that RFC is attached to this declaration as **Exhibit 4.**

15. Based on the business records for the RFC Editor and the RFC Editor's course of conduct in publishing RFCs, I have determined that the publication date of RFC 4302 (IP Authentication Header) was no later than December, 2005, at which time it was

reasonably accessible to the public either on the RFC Editor website or via FTP from a repository. An announcement of its publication also would have been sent out to subscribers within 24 hours of its publication. A copy of that RFC is attached to this declaration as **Exhibit 5**.

16. Based on the business records for the RFC Editor and the RFC Editor's course of conduct in publishing RFCs, I have determined that the publication date of RFC 6101 (The Secure Sockets Layer (SSL) Protocol Version 3.0) was no later than August, 2011, at which time it was reasonably accessible to the public either on the RFC Editor website or via FTP from a repository. An announcement of its publication also would have been sent out to subscribers within 24 hours of its publication. A copy of that RFC is attached to this declaration as **Exhibit 6.**

[REMAINDER OF PAGE LEFT INTENTIONALLY BLANK]

PURSUANT TO SECTION 1746 OF TITLE 28 OF UNITED STATES CODE, I DECLARE UNDER PENALTY OF PERJURY UNDER THE LAWS OF THE UNITED STATES OF AMERICA THAT THE FOREGOING IS TRUE AND CORRECT AND THAT THE FOREGOING IS BASED UPON PERSONAL KNOWLEDGE AND INFORMATION AND IS BELIEVED TO BE TRUE.

Date: 12 December 2024 By: _____
A. Jean Mahoney

Exhibit 1

Network Working Group
Request for Comments: 4251
Category: Standards Track

T. Ylonen
SSH Communications Security Corp
C. Lonvick, Ed.
Cisco Systems, Inc.

January 2006

The Secure Shell (SSH) Protocol Architecture

Status of This Memo

Copyright Notice

Abstract

The Secure Shell (SSH) Protocol is a protocol for secure remote login and other secure network services over an insecure network. This document describes the architecture of the SSH protocol, as well as the notation and terminology used in SSH protocol documents. It also discusses the SSH algorithm naming system that allows local extensions. The SSH protocol consists of three major components: The Transport Layer Protocol provides server authentication, confidentiality, and integrity with perfect forward secrecy. The User Authentication Protocol authenticates the client to the server. The Connection Protocol multiplexes the encrypted tunnel into several logical channels. Details of these protocols are described in separate documents.

Table of Contents

1. Introduction

Secure Shell (SSH) is a protocol for secure remote login and other secure network services over an insecure network. It consists of three major components:

- The Transport Layer Protocol [SSH-TRANS] provides server authentication, confidentiality, and integrity. It may optionally also provide compression. The transport layer will typically be run over a TCP/IP connection, but might also be used on top of any other reliable data stream.

- The User Authentication Protocol [SSH-USERAUTH] authenticates the client-side user to the server. It runs over the transport layer protocol.

- The Connection Protocol [SSH-CONNECT] multiplexes the encrypted tunnel into several logical channels. It runs over the user authentication protocol.

The client sends a service request once a secure transport layer connection has been established. A second service request is sent after user authentication is complete. This allows new protocols to be defined and coexist with the protocols listed above.

The connection protocol provides channels that can be used for a wide range of purposes. Standard methods are provided for setting up secure interactive shell sessions and for forwarding ("tunneling") arbitrary TCP/IP ports and X11 connections.

2. Contributors

The major original contributors of this set of documents have been: Tatu Ylonen, Tero Kivinen, Timo J. Rinne, Sami Lehtinen (all of SSH Communications Security Corp), and Markku-Juhani O. Saarinen (University of Jyvaskyla). Darren Moffat was the original editor of this set of documents and also made very substantial contributions.

Many people contributed to the development of this document over the years. People who should be acknowledged include Mats Andersson, Ben Harris, Bill Sommerfeld, Brent McClure, Niels Moller, Damien Miller, Derek Fawcus, Frank Cusack, Heikki Nousiainen, Jakob Schlyter, Jeff Van Dyke, Jeffrey Altman, Jeffrey Hutzelman, Jon Bright, Joseph Galbraith, Ken Hornstein, Markus Friedl, Martin Forssen, Nicolas Williams, Niels Provos, Perry Metzger, Peter Gutmann, Simon Josefsson, Simon Tatham, Wei Dai, Denis Bider, der Mouse, and Tadayoshi Kohno. Listing their names here does not mean that they endorse this document, but that they have contributed to it.

3. Conventions Used in This Document

All documents related to the SSH protocols shall use the keywords "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" to describe requirements. These keywords are to be interpreted as described in [RFC2119].

The keywords "PRIVATE USE", "HIERARCHICAL ALLOCATION", "FIRST COME FIRST SERVED", "EXPERT REVIEW", "SPECIFICATION REQUIRED", "IESG APPROVAL", "IETF CONSENSUS", and "STANDARDS ACTION" that appear in this document when used to describe namespace allocation are to be interpreted as described in [RFC2434].

Protocol fields and possible values to fill them are defined in this set of documents. Protocol fields will be defined in the message definitions. As an example, SSH_MSG_CHANNEL_DATA is defined as follows.

```
byte      SSH_MSG_CHANNEL_DATA
uint32    recipient channel
string    data
```

Throughout these documents, when the fields are referenced, they will appear within single quotes. When values to fill those fields are referenced, they will appear within double quotes. Using the above example, possible values for 'data' are "foo" and "bar".

4. Architecture

4.1. Host Keys

Each server host SHOULD have a host key. Hosts MAY have multiple host keys using multiple different algorithms. Multiple hosts MAY share the same host key. If a host has keys at all, it MUST have at least one key that uses each REQUIRED public key algorithm (DSS [FIPS-186-2]).

The server host key is used during key exchange to verify that the client is really talking to the correct server. For this to be possible, the client must have a priori knowledge of the server's public host key.

Two different trust models can be used:

- The client has a local database that associates each host name (as typed by the user) with the corresponding public host key. This method requires no centrally administered infrastructure, and no

third-party coordination. The downside is that the database of name-to-key associations may become burdensome to maintain.

- The host name-to-key association is certified by a trusted certification authority (CA). The client only knows the CA root key, and can verify the validity of all host keys certified by accepted CAs.

The second alternative eases the maintenance problem, since ideally only a single CA key needs to be securely stored on the client. On the other hand, each host key must be appropriately certified by a central authority before authorization is possible. Also, a lot of trust is placed on the central infrastructure.

The protocol provides the option that the server name - host key association is not checked when connecting to the host for the first time. This allows communication without prior communication of host keys or certification. The connection still provides protection against passive listening; however, it becomes vulnerable to active man-in-the-middle attacks. Implementations SHOULD NOT normally allow such connections by default, as they pose a potential security problem. However, as there is no widely deployed key infrastructure available on the Internet at the time of this writing, this option makes the protocol much more usable during the transition time until such an infrastructure emerges, while still providing a much higher level of security than that offered by older solutions (e.g., telnet [RFC0854] and rlogin [RFC1282]).

Implementations SHOULD try to make the best effort to check host keys. An example of a possible strategy is to only accept a host key without checking the first time a host is connected, save the key in a local database, and compare against that key on all future connections to that host.

Implementations MAY provide additional methods for verifying the correctness of host keys, e.g., a hexadecimal fingerprint derived from the SHA-1 hash [FIPS-180-2] of the public key. Such fingerprints can easily be verified by using telephone or other external communication channels.

All implementations SHOULD provide an option not to accept host keys that cannot be verified.

The members of this Working Group believe that 'ease of use' is critical to end-user acceptance of security solutions, and no improvement in security is gained if the new solutions are not used. Thus, providing the option not to check the server host key is

believed to improve the overall security of the Internet, even though it reduces the security of the protocol in configurations where it is allowed.

4.2. Extensibility

We believe that the protocol will evolve over time, and some organizations will want to use their own encryption, authentication, and/or key exchange methods. Central registration of all extensions is cumbersome, especially for experimental or classified features. On the other hand, having no central registration leads to conflicts in method identifiers, making interoperability difficult.

We have chosen to identify algorithms, methods, formats, and extension protocols with textual names that are of a specific format. DNS names are used to create local namespaces where experimental or classified extensions can be defined without fear of conflicts with other implementations.

One design goal has been to keep the base protocol as simple as possible, and to require as few algorithms as possible. However, all implementations MUST support a minimal set of algorithms to ensure interoperability (this does not imply that the local policy on all hosts would necessarily allow these algorithms). The mandatory algorithms are specified in the relevant protocol documents.

Additional algorithms, methods, formats, and extension protocols can be defined in separate documents. See Section 6, Algorithm Naming, for more information.

4.3. Policy Issues

The protocol allows full negotiation of encryption, integrity, key exchange, compression, and public key algorithms and formats. Encryption, integrity, public key, and compression algorithms can be different for each direction.

The following policy issues SHOULD be addressed in the configuration mechanisms of each implementation:

- Encryption, integrity, and compression algorithms, separately for each direction. The policy MUST specify which is the preferred algorithm (e.g., the first algorithm listed in each category).

- Public key algorithms and key exchange method to be used for host authentication. The existence of trusted host keys for different public key algorithms also affects this choice.

- The authentication methods that are to be required by the server for each user. The server's policy MAY require multiple authentication for some or all users. The required algorithms MAY depend on the location from where the user is trying to gain access.

- The operations that the user is allowed to perform using the connection protocol. Some issues are related to security; for example, the policy SHOULD NOT allow the server to start sessions or run commands on the client machine, and MUST NOT allow connections to the authentication agent unless forwarding such connections has been requested. Other issues, such as which TCP/IP ports can be forwarded and by whom, are clearly issues of local policy. Many of these issues may involve traversing or bypassing firewalls, and are interrelated with the local security policy.

4.4. Security Properties

The primary goal of the SSH protocol is to improve security on the Internet. It attempts to do this in a way that is easy to deploy, even at the cost of absolute security.

- All encryption, integrity, and public key algorithms used are well-known, well-established algorithms.

- All algorithms are used with cryptographically sound key sizes that are believed to provide protection against even the strongest cryptanalytic attacks for decades.

- All algorithms are negotiated, and in case some algorithm is broken, it is easy to switch to some other algorithm without modifying the base protocol.

Specific concessions were made to make widespread, fast deployment easier. The particular case where this comes up is verifying that the server host key really belongs to the desired host; the protocol allows the verification to be left out, but this is NOT RECOMMENDED. This is believed to significantly improve usability in the short term, until widespread Internet public key infrastructures emerge.

4.5. Localization and Character Set Support

For the most part, the SSH protocols do not directly pass text that would be displayed to the user. However, there are some places where such data might be passed. When applicable, the character set for

the data MUST be explicitly specified. In most places, ISO-10646 UTF-8 encoding is used [RFC3629]. When applicable, a field is also provided for a language tag [RFC3066].

One big issue is the character set of the interactive session. There is no clear solution, as different applications may display data in different formats. Different types of terminal emulation may also be employed in the client, and the character set to be used is effectively determined by the terminal emulation. Thus, no place is provided for directly specifying the character set or encoding for terminal session data. However, the terminal emulation type (e.g., "vt100") is transmitted to the remote site, and it implicitly specifies the character set and encoding. Applications typically use the terminal type to determine what character set they use, or the character set is determined using some external means. The terminal emulation may also allow configuring the default character set. In any case, the character set for the terminal session is considered primarily a client local issue.

Internal names used to identify algorithms or protocols are normally never displayed to users, and must be in US-ASCII.

The client and server user names are inherently constrained by what the server is prepared to accept. They might, however, occasionally be displayed in logs, reports, etc. They MUST be encoded using ISO 10646 UTF-8, but other encodings may be required in some cases. It is up to the server to decide how to map user names to accepted user names. Straight bit-wise, binary comparison is RECOMMENDED.

For localization purposes, the protocol attempts to minimize the number of textual messages transmitted. When present, such messages typically relate to errors, debugging information, or some externally configured data. For data that is normally displayed, it SHOULD be possible to fetch a localized message instead of the transmitted message by using a numerical code. The remaining messages SHOULD be configurable.

5. Data Type Representations Used in the SSH Protocols

byte

A byte represents an arbitrary 8-bit value (octet). Fixed length data is sometimes represented as an array of bytes, written byte[n], where n is the number of bytes in the array.

boolean

A boolean value is stored as a single byte. The value 0 represents FALSE, and the value 1 represents TRUE. All non-zero values MUST be interpreted as TRUE; however, applications MUST NOT store values other than 0 and 1.

uint32

Represents a 32-bit unsigned integer. Stored as four bytes in the order of decreasing significance (network byte order). For example: the value 699921578 (0x29b7f4aa) is stored as 29 b7 f4 aa.

uint64

Represents a 64-bit unsigned integer. Stored as eight bytes in the order of decreasing significance (network byte order).

string

Arbitrary length binary string. Strings are allowed to contain arbitrary binary data, including null characters and 8-bit characters. They are stored as a uint32 containing its length (number of bytes that follow) and zero (= empty string) or more bytes that are the value of the string. Terminating null characters are not used.

Strings are also used to store text. In that case, US-ASCII is used for internal names, and ISO-10646 UTF-8 for text that might be displayed to the user. The terminating null character SHOULD NOT normally be stored in the string. For example: the US-ASCII string "testing" is represented as 00 00 00 07 t e s t i n g. The UTF-8 mapping does not alter the encoding of US-ASCII characters.

mpint

Represents multiple precision integers in two’s complement format, stored as a string, 8 bits per byte, MSB first. Negative numbers have the value 1 as the most significant bit of the first byte of the data partition. If the most significant bit would be set for a positive number, the number MUST be preceded by a zero byte. Unnecessary leading bytes with the value 0 or 255 MUST NOT be included. The value zero MUST be stored as a string with zero bytes of data.

By convention, a number that is used in modular computations in Z_n SHOULD be represented in the range 0 <= x < n.

Examples:

value (hex)	representation (hex)
0	00 00 00 00
9a378f9b2e332a7	00 00 00 08 09 a3 78 f9 b2 e3 32 a7
80	00 00 00 02 00 80
-1234	00 00 00 02 ed cc
-deadbeef	00 00 00 05 ff 21 52 41 11

name-list

A string containing a comma-separated list of names. A name-list is represented as a uint32 containing its length (number of bytes that follow) followed by a comma-separated list of zero or more names. A name MUST have a non-zero length, and it MUST NOT contain a comma (","). As this is a list of names, all of the elements contained are names and MUST be in US-ASCII. Context may impose additional restrictions on the names. For example, the names in a name-list may have to be a list of valid algorithm identifiers (see Section 6 below), or a list of [RFC3066] language tags. The order of the names in a name-list may or may not be significant. Again, this depends on the context in which the list is used. Terminating null characters MUST NOT be used, neither for the individual names, nor for the list as a whole.

Examples:

value	representation (hex)
(), the empty name-list	00 00 00 00
("zlib")	00 00 00 04 7a 6c 69 62
("zlib,none")	00 00 00 09 7a 6c 69 62 2c 6e 6f 6e 65

6. Algorithm and Method Naming

The SSH protocols refer to particular hash, encryption, integrity, compression, and key exchange algorithms or methods by name. There are some standard algorithms and methods that all implementations MUST support. There are also algorithms and methods that are defined in the protocol specification, but are OPTIONAL. Furthermore, it is expected that some organizations will want to use their own algorithms or methods.

In this protocol, all algorithm and method identifiers MUST be printable US-ASCII, non-empty strings no longer than 64 characters. Names MUST be case-sensitive.

There are two formats for algorithm and method names:

- Names that do not contain an at-sign ("@") are reserved to be assigned by IETF CONSENSUS. Examples include "3des-cbc", "sha-1", "hmac-sha1", and "zlib" (the doublequotes are not part of the name). Names of this format are only valid if they are first registered with the IANA. Registered names MUST NOT contain an at-sign ("@"), comma (","), whitespace, control characters (ASCII codes 32 or less), or the ASCII code 127 (DEL). Names are case-sensitive, and MUST NOT be longer than 64 characters.

- Anyone can define additional algorithms or methods by using names in the format name@domainname, e.g., "ourcipher-cbc@example.com". The format of the part preceding the at-sign is not specified; however, these names MUST be printable US-ASCII strings, and MUST NOT contain the comma character (","), whitespace, control characters (ASCII codes 32 or less), or the ASCII code 127 (DEL). They MUST have only a single at-sign in them. The part following the at-sign MUST be a valid, fully qualified domain name [RFC1034] controlled by the person or organization defining the name. Names are case-sensitive, and MUST NOT be longer than 64 characters. It is up to each domain how it manages its local namespace. It should be noted that these names resemble STD 11 [RFC0822] email addresses. This is purely coincidental and has nothing to do with STD 11 [RFC0822].

7. Message Numbers

SSH packets have message numbers in the range 1 to 255. These numbers have been allocated as follows:

Transport layer protocol:

1 to 19	Transport layer generic (e.g., disconnect, ignore, debug, etc.)
20 to 29	Algorithm negotiation
30 to 49	Key exchange method specific (numbers can be reused for different authentication methods)

User authentication protocol:

50 to 59	User authentication generic
60 to 79	User authentication method specific (numbers can be reused for different authentication methods)

Connection protocol:

```
80 to 89   Connection protocol generic
90 to 127  Channel related messages
```

Reserved for client protocols:

```
128 to 191 Reserved
```

Local extensions:

```
192 to 255 Local extensions
```

8. IANA Considerations

This document is part of a set. The instructions for the IANA for the SSH protocol, as defined in this document, [SSH-USERAUTH], [SSH-TRANS], and [SSH-CONNECT], are detailed in [SSH-NUMBERS]. The following is a brief summary for convenience, but note well that [SSH-NUMBERS] contains the actual instructions to the IANA, which may be superseded in the future.

Allocation of the following types of names in the SSH protocols is assigned by IETF consensus:

- Service Names
  - Authentication Methods
  - Connection Protocol Channel Names
  - Connection Protocol Global Request Names
  - Connection Protocol Channel Request Names
- Key Exchange Method Names
- Assigned Algorithm Names
  - Encryption Algorithm Names
  - MAC Algorithm Names
  - Public Key Algorithm Names
  - Compression Algorithm Names

These names MUST be printable US-ASCII strings, and MUST NOT contain the characters at-sign ("@"), comma (","), whitespace, control characters (ASCII codes 32 or less), or the ASCII code 127 (DEL). Names are case-sensitive, and MUST NOT be longer than 64 characters.

Names with the at-sign ("@") are locally defined extensions and are not controlled by the IANA.

Each category of names listed above has a separate namespace. However, using the same name in multiple categories SHOULD be avoided to minimize confusion.

Message numbers (see Section 7) in the range of 0 to 191 are allocated via IETF CONSENSUS, as described in [RFC2434]. Message numbers in the 192 to 255 range (local extensions) are reserved for PRIVATE USE, also as described in [RFC2434].

9. Security Considerations

In order to make the entire body of Security Considerations more accessible, Security Considerations for the transport, authentication, and connection documents have been gathered here.

The transport protocol [SSH-TRANS] provides a confidential channel over an insecure network. It performs server host authentication, key exchange, encryption, and integrity protection. It also derives a unique session id that may be used by higher-level protocols.

The authentication protocol [SSH-USERAUTH] provides a suite of mechanisms that can be used to authenticate the client user to the server. Individual mechanisms specified in the authentication protocol use the session id provided by the transport protocol and/or depend on the security and integrity guarantees of the transport protocol.

The connection protocol [SSH-CONNECT] specifies a mechanism to multiplex multiple streams (channels) of data over the confidential and authenticated transport. It also specifies channels for accessing an interactive shell, for proxy-forwarding various external protocols over the secure transport (including arbitrary TCP/IP protocols), and for accessing secure subsystems on the server host.

9.1. Pseudo-Random Number Generation

This protocol binds each session key to the session by including random, session specific data in the hash used to produce session keys. Special care should be taken to ensure that all of the random numbers are of good quality. If the random data here (e.g., Diffie-Hellman (DH) parameters) are pseudo-random, then the pseudo-random number generator should be cryptographically secure (i.e., its next output not easily guessed even when knowing all previous outputs) and, furthermore, proper entropy needs to be added to the pseudo-random number generator. [RFC4086] offers suggestions for sources of random numbers and entropy. Implementers should note the importance of entropy and the well-meant, anecdotal warning about the difficulty in properly implementing pseudo-random number generating functions.

The amount of entropy available to a given client or server may sometimes be less than what is required. In this case, one must either resort to pseudo-random number generation regardless of insufficient entropy or refuse to run the protocol. The latter is preferable.

9.2. Control Character Filtering

When displaying text to a user, such as error or debug messages, the client software SHOULD replace any control characters (except tab, carriage return, and newline) with safe sequences to avoid attacks by sending terminal control characters.

9.3. Transport

9.3.1. Confidentiality

It is beyond the scope of this document and the Secure Shell Working Group to analyze or recommend specific ciphers other than the ones that have been established and accepted within the industry. At the time of this writing, commonly used ciphers include 3DES, ARCFOUR, twofish, serpent, and blowfish. AES has been published by The US Federal Information Processing Standards as [FIPS-197], and the cryptographic community has accepted AES as well. As always, implementers and users should check current literature to ensure that no recent vulnerabilities have been found in ciphers used within products. Implementers should also check to see which ciphers are considered to be relatively stronger than others and should recommend their use to users over relatively weaker ciphers. It would be considered good form for an implementation to politely and unobtrusively notify a user that a stronger cipher is available and should be used when a weaker one is actively chosen.

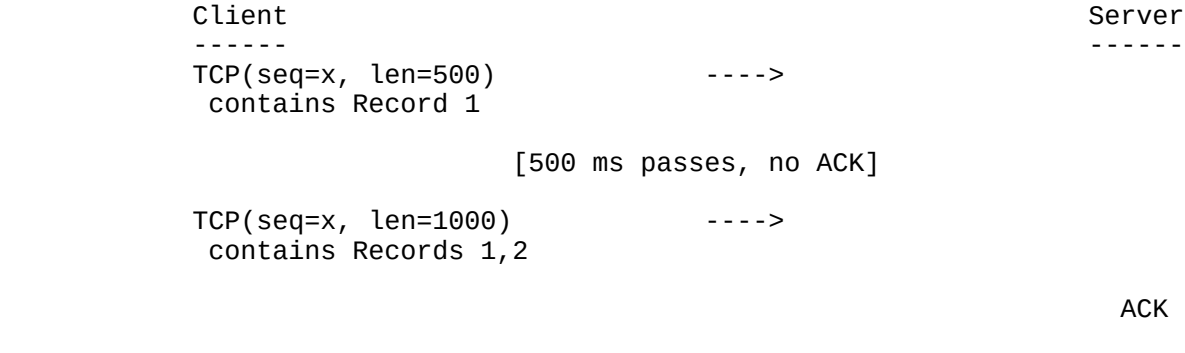
The "none" cipher is provided for debugging and SHOULD NOT be used except for that purpose. Its cryptographic properties are sufficiently described in [RFC2410], which will show that its use does not meet the intent of this protocol.

The relative merits of these and other ciphers may also be found in current literature. Two references that may provide information on the subject are [SCHNEIER] and [KAUFMAN]. Both of these describe the CBC mode of operation of certain ciphers and the weakness of this scheme. Essentially, this mode is theoretically vulnerable to chosen cipher-text attacks because of the high predictability of the start of packet sequence. However, this attack is deemed difficult and not considered fully practicable, especially if relatively long block sizes are used.

Additionally, another CBC mode attack may be mitigated through the insertion of packets containing SSH_MSG_IGNORE. Without this technique, a specific attack may be successful. For this attack (commonly known as the Rogaway attack [ROGAWAY], [DAI], [BELLARE]) to work, the attacker would need to know the Initialization Vector (IV) of the next block that is going to be encrypted. In CBC mode that is the output of the encryption of the previous block. If the attacker does not have any way to see the packet yet (i.e., it is in the internal buffers of the SSH implementation or even in the kernel), then this attack will not work. If the last packet has been sent out to the network (i.e., the attacker has access to it), then he can use the attack.

In the optimal case, an implementer would need to add an extra packet only if the packet has been sent out onto the network and there are no other packets waiting for transmission. Implementers may wish to check if there are any unsent packets awaiting transmission; unfortunately, it is not normally easy to obtain this information from the kernel or buffers. If there are no unsent packets, then a packet containing SSH_MSG_IGNORE SHOULD be sent. If a new packet is added to the stream every time the attacker knows the IV that is supposed to be used for the next packet, then the attacker will not be able to guess the correct IV, thus the attack will never be successful.

As an example, consider the following case:



1. The Nagle algorithm + TCP retransmits mean that the two records get coalesced into a single TCP segment.

2. Record 2 is not at the beginning of the TCP segment and never will be because it gets ACKed.

3. Yet, the attack is possible because Record 1 has already been seen.

As this example indicates, it is unsafe to use the existence of unflushed data in the TCP buffers proper as a guide to whether an empty packet is needed, since when the second write() is performed the buffers will contain the un-ACKed Record 1.

On the other hand, it is perfectly safe to have the following situation:

```
Client                                                  Server
------                                                  ------
TCP(seq=x, len=500)             ---->
   contains SSH_MSG_IGNORE

TCP(seq=y, len=500)             ---->
   contains Data
```

Provided that the IV for the second SSH Record is fixed after the data for the Data packet is determined, then the following should be performed:

```
read from user
encrypt null packet
encrypt data packet
```

9.3.2. Data Integrity

This protocol does allow the Data Integrity mechanism to be disabled. Implementers SHOULD be wary of exposing this feature for any purpose other than debugging. Users and administrators SHOULD be explicitly warned anytime the "none" MAC is enabled.

So long as the "none" MAC is not used, this protocol provides data integrity.

Because MACs use a 32-bit sequence number, they might start to leak information after 2**32 packets have been sent. However, following the rekeying recommendations should prevent this attack. The transport protocol [SSH-TRANS] recommends rekeying after one gigabyte of data, and the smallest possible packet is 16 bytes. Therefore, rekeying SHOULD happen after 2**28 packets at the very most.

9.3.3. Replay

The use of a MAC other than "none" provides integrity and authentication. In addition, the transport protocol provides a unique session identifier (bound in part to pseudo-random data that is part of the algorithm and key exchange process) that can be used by higher level protocols to bind data to a given session and prevent

replay of data from prior sessions. For example, the authentication protocol ([SSH-USERAUTH]) uses this to prevent replay of signatures from previous sessions. Because public key authentication exchanges are cryptographically bound to the session (i.e., to the initial key exchange), they cannot be successfully replayed in other sessions. Note that the session id can be made public without harming the security of the protocol.

If two sessions have the same session id (hash of key exchanges), then packets from one can be replayed against the other. It must be stressed that the chances of such an occurrence are, needless to say, minimal when using modern cryptographic methods. This is all the more true when specifying larger hash function outputs and DH parameters.

Replay detection using monotonically increasing sequence numbers as input to the MAC, or HMAC in some cases, is described in [RFC2085], [RFC2246], [RFC2743], [RFC1964], [RFC2025], and [RFC4120]. The underlying construct is discussed in [RFC2104]. Essentially, a different sequence number in each packet ensures that at least this one input to the MAC function will be unique and will provide a nonrecurring MAC output that is not predictable to an attacker. If the session stays active long enough, however, this sequence number will wrap. This event may provide an attacker an opportunity to replay a previously recorded packet with an identical sequence number but only if the peers have not rekeyed since the transmission of the first packet with that sequence number. If the peers have rekeyed, then the replay will be detected since the MAC check will fail. For this reason, it must be emphasized that peers MUST rekey before a wrap of the sequence numbers. Naturally, if an attacker does attempt to replay a captured packet before the peers have rekeyed, then the receiver of the duplicate packet will not be able to validate the MAC and it will be discarded. The reason that the MAC will fail is because the receiver will formulate a MAC based upon the packet contents, the shared secret, and the expected sequence number. Since the replayed packet will not be using that expected sequence number (the sequence number of the replayed packet will have already been passed by the receiver), the calculated MAC will not match the MAC received with the packet.

9.3.4. Man-in-the-middle

This protocol makes no assumptions or provisions for an infrastructure or means for distributing the public keys of hosts. It is expected that this protocol will sometimes be used without first verifying the association between the server host key and the server host name. Such usage is vulnerable to man-in-the-middle attacks. This section describes this and encourages administrators

and users to understand the importance of verifying this association before any session is initiated.

There are three cases of man-in-the-middle attacks to consider. The first is where an attacker places a device between the client and the server before the session is initiated. In this case, the attack device is trying to mimic the legitimate server and will offer its public key to the client when the client initiates a session. If it were to offer the public key of the server, then it would not be able to decrypt or sign the transmissions between the legitimate server and the client unless it also had access to the private key of the host. The attack device will also, simultaneously to this, initiate a session to the legitimate server, masquerading itself as the client. If the public key of the server had been securely distributed to the client prior to that session initiation, the key offered to the client by the attack device will not match the key stored on the client. In that case, the user SHOULD be given a warning that the offered host key does not match the host key cached on the client. As described in Section 4.1, the user may be free to accept the new key and continue the session. It is RECOMMENDED that the warning provide sufficient information to the user of the client device so the user may make an informed decision. If the user chooses to continue the session with the stored public key of the server (not the public key offered at the start of the session), then the session-specific data between the attacker and server will be different between the client-to-attacker session and the attacker-to-server sessions due to the randomness discussed above. From this, the attacker will not be able to make this attack work since the attacker will not be able to correctly sign packets containing this session-specific data from the server, since he does not have the private key of that server.

The second case that should be considered is similar to the first case in that it also happens at the time of connection, but this case points out the need for the secure distribution of server public keys. If the server public keys are not securely distributed, then the client cannot know if it is talking to the intended server. An attacker may use social engineering techniques to pass off server keys to unsuspecting users and may then place a man-in-the-middle attack device between the legitimate server and the clients. If this is allowed to happen, then the clients will form client-to-attacker sessions, and the attacker will form attacker-to-server sessions and will be able to monitor and manipulate all of the traffic between the clients and the legitimate servers. Server administrators are encouraged to make host key fingerprints available for checking by some means whose security does not rely on the integrity of the actual host keys. Possible mechanisms are discussed in Section 4.1 and may also include secured Web pages, physical pieces of paper,

etc. Implementers SHOULD provide recommendations on how best to do this with their implementation. Because the protocol is extensible, future extensions to the protocol may provide better mechanisms for dealing with the need to know the server's host key before connecting. For example, making the host key fingerprint available through a secure DNS lookup, or using Kerberos ([RFC4120]) over GSS-API ([RFC1964]) during key exchange to authenticate the server are possibilities.

In the third man-in-the-middle case, attackers may attempt to manipulate packets in transit between peers after the session has been established. As described in Section 9.3.3, a successful attack of this nature is very improbable. As in Section 9.3.3, this reasoning does assume that the MAC is secure and that it is infeasible to construct inputs to a MAC algorithm to give a known output. This is discussed in much greater detail in Section 6 of [RFC2104]. If the MAC algorithm has a vulnerability or is weak enough, then the attacker may be able to specify certain inputs to yield a known MAC. With that, they may be able to alter the contents of a packet in transit. Alternatively, the attacker may be able to exploit the algorithm vulnerability or weakness to find the shared secret by reviewing the MACs from captured packets. In either of those cases, an attacker could construct a packet or packets that could be inserted into an SSH stream. To prevent this, implementers are encouraged to utilize commonly accepted MAC algorithms, and administrators are encouraged to watch current literature and discussions of cryptography to ensure that they are not using a MAC algorithm that has a recently found vulnerability or weakness.

In summary, the use of this protocol without a reliable association of the binding between a host and its host keys is inherently insecure and is NOT RECOMMENDED. However, it may be necessary in non-security-critical environments, and will still provide protection against passive attacks. Implementers of protocols and applications running on top of this protocol should keep this possibility in mind.

9.3.5. Denial of Service

This protocol is designed to be used over a reliable transport. If transmission errors or message manipulation occur, the connection is closed. The connection SHOULD be re-established if this occurs. Denial of service attacks of this type (wire cutter) are almost impossible to avoid.

In addition, this protocol is vulnerable to denial of service attacks because an attacker can force the server to go through the CPU and memory intensive tasks of connection setup and key exchange without authenticating. Implementers SHOULD provide features that make this

more difficult, for example, only allowing connections from a subset of clients known to have valid users.

9.3.6. Covert Channels

The protocol was not designed to eliminate covert channels. For example, the padding, SSH_MSG_IGNORE messages, and several other places in the protocol can be used to pass covert information, and the recipient has no reliable way of verifying whether such information is being sent.

9.3.7. Forward Secrecy

It should be noted that the Diffie-Hellman key exchanges may provide perfect forward secrecy (PFS). PFS is essentially defined as the cryptographic property of a key-establishment protocol in which the compromise of a session key or long-term private key after a given session does not cause the compromise of any earlier session [ANSI-T1.523-2001]. SSH sessions resulting from a key exchange using the diffie-hellman methods described in the section Diffie-Hellman Key Exchange of [SSH-TRANS] (including "diffie-hellman-group1-sha1" and "diffie-hellman-group14-sha1") are secure even if private keying/authentication material is later revealed, but not if the session keys are revealed. So, given this definition of PFS, SSH does have PFS. However, this property is not commuted to any of the applications or protocols using SSH as a transport. The transport layer of SSH provides confidentiality for password authentication and other methods that rely on secret data.

Of course, if the DH private parameters for the client and server are revealed, then the session key is revealed, but these items can be thrown away after the key exchange completes. It’s worth pointing out that these items should not be allowed to end up on swap space and that they should be erased from memory as soon as the key exchange completes.

9.3.8. Ordering of Key Exchange Methods

As stated in the section on Algorithm Negotiation of [SSH-TRANS], each device will send a list of preferred methods for key exchange. The most-preferred method is the first in the list. It is RECOMMENDED that the algorithms be sorted by cryptographic strength, strongest first. Some additional guidance for this is given in [RFC3766].

9.3.9. Traffic Analysis

Passive monitoring of any protocol may give an attacker some information about the session, the user, or protocol specific information that they would otherwise not be able to garner. For example, it has been shown that traffic analysis of an SSH session can yield information about the length of the password - [Openwall] and [USENIX]. Implementers should use the SSH_MSG_IGNORE packet, along with the inclusion of random lengths of padding, to thwart attempts at traffic analysis. Other methods may also be found and implemented.

9.4. Authentication Protocol

The purpose of this protocol is to perform client user authentication. It assumes that this runs over a secure transport layer protocol, which has already authenticated the server machine, established an encrypted communications channel, and computed a unique session identifier for this session.

Several authentication methods with different security characteristics are allowed. It is up to the server's local policy to decide which methods (or combinations of methods) it is willing to accept for each user. Authentication is no stronger than the weakest combination allowed.

The server may go into a sleep period after repeated unsuccessful authentication attempts to make key search more difficult for attackers. Care should be taken so that this doesn't become a self-denial of service vector.

9.4.1. Weak Transport

If the transport layer does not provide confidentiality, authentication methods that rely on secret data SHOULD be disabled. If it does not provide strong integrity protection, requests to change authentication data (e.g., a password change) SHOULD be disabled to prevent an attacker from modifying the ciphertext without being noticed, or rendering the new authentication data unusable (denial of service).

The assumption stated above, that the Authentication Protocol only runs over a secure transport that has previously authenticated the server, is very important to note. People deploying SSH are reminded of the consequences of man-in-the-middle attacks if the client does not have a very strong a priori association of the server with the host key of that server. Specifically, for the case of the Authentication Protocol, the client may form a session to a man-in-

the-middle attack device and divulge user credentials such as their username and password. Even in the cases of authentication where no user credentials are divulged, an attacker may still gain information they shouldn't have by capturing key-strokes in much the same way that a honeypot works.

9.4.2. Debug Messages

Special care should be taken when designing debug messages. These messages may reveal surprising amounts of information about the host if not properly designed. Debug messages can be disabled (during user authentication phase) if high security is required. Administrators of host machines should make all attempts to compartmentalize all event notification messages and protect them from unwarranted observation. Developers should be aware of the sensitive nature of some of the normal event and debug messages, and may want to provide guidance to administrators on ways to keep this information away from unauthorized people. Developers should consider minimizing the amount of sensitive information obtainable by users during the authentication phase, in accordance with the local policies. For this reason, it is RECOMMENDED that debug messages be initially disabled at the time of deployment and require an active decision by an administrator to allow them to be enabled. It is also RECOMMENDED that a message expressing this concern be presented to the administrator of a system when the action is taken to enable debugging messages.

9.4.3. Local Security Policy

The implementer MUST ensure that the credentials provided validate the professed user and also MUST ensure that the local policy of the server permits the user the access requested. In particular, because of the flexible nature of the SSH connection protocol, it may not be possible to determine the local security policy, if any, that should apply at the time of authentication because the kind of service being requested is not clear at that instant. For example, local policy might allow a user to access files on the server, but not start an interactive shell. However, during the authentication protocol, it is not known whether the user will be accessing files, attempting to use an interactive shell, or even both. In any event, where local security policy for the server host exists, it MUST be applied and enforced correctly.

Implementers are encouraged to provide a default local policy and make its parameters known to administrators and users. At the discretion of the implementers, this default policy may be along the lines of anything-goes where there are no restrictions placed upon users, or it may be along the lines of excessively-restrictive, in

which case, the administrators will have to actively make changes to the initial default parameters to meet their needs. Alternatively, it may be some attempt at providing something practical and immediately useful to the administrators of the system so they don't have to put in much effort to get SSH working. Whatever choice is made must be applied and enforced as required above.

9.4.4 Public Key Authentication

The use of public key authentication assumes that the client host has not been compromised. It also assumes that the private key of the server host has not been compromised.

This risk can be mitigated by the use of passphrases on private keys; however, this is not an enforceable policy. The use of smartcards, or other technology to make passphrases an enforceable policy is suggested.

The server could require both password and public key authentication; however, this requires the client to expose its password to the server (see the section on Password Authentication below.)

9.4.5. Password Authentication

The password mechanism, as specified in the authentication protocol, assumes that the server has not been compromised. If the server has been compromised, using password authentication will reveal a valid username/password combination to the attacker, which may lead to further compromises.

This vulnerability can be mitigated by using an alternative form of authentication. For example, public key authentication makes no assumptions about security on the server.

9.4.6. Host-Based Authentication

Host-based authentication assumes that the client has not been compromised. There are no mitigating strategies, other than to use host-based authentication in combination with another authentication method.

9.5. Connection Protocol

9.5.1. End Point Security

End point security is assumed by the connection protocol. If the server has been compromised, any terminal sessions, port forwarding, or systems accessed on the host are compromised. There are no mitigating factors for this.

If the client has been compromised, and the server fails to stop the attacker at the authentication protocol, all services exposed (either as subsystems or through forwarding) will be vulnerable to attack. Implementers SHOULD provide mechanisms for administrators to control which services are exposed to limit the vulnerability of other services. These controls might include controlling which machines and ports can be targeted in port-forwarding operations, which users are allowed to use interactive shell facilities, or which users are allowed to use exposed subsystems.

9.5.2. Proxy Forwarding

The SSH connection protocol allows for proxy forwarding of other protocols such as SMTP, POP3, and HTTP. This may be a concern for network administrators who wish to control the access of certain applications by users located outside of their physical location. Essentially, the forwarding of these protocols may violate site-specific security policies, as they may be undetectably tunneled through a firewall. Implementers SHOULD provide an administrative mechanism to control the proxy forwarding functionality so that site-specific security policies may be upheld.

In addition, a reverse proxy forwarding functionality is available, which, again, can be used to bypass firewall controls.

As indicated above, end-point security is assumed during proxy forwarding operations. Failure of end-point security will compromise all data passed over proxy forwarding.

9.5.3. X11 Forwarding

Another form of proxy forwarding provided by the SSH connection protocol is the forwarding of the X11 protocol. If end-point security has been compromised, X11 forwarding may allow attacks against the X11 server. Users and administrators should, as a matter of course, use appropriate X11 security mechanisms to prevent unauthorized use of the X11 server. Implementers, administrators, and users who wish to further explore the security mechanisms of X11 are invited to read [SCHEIFLER] and analyze previously reported

problems with the interactions between SSH forwarding and X11 in CERT vulnerabilities VU#363181 and VU#118892 [CERT].

X11 display forwarding with SSH, by itself, is not sufficient to correct well known problems with X11 security [VENEMA]. However, X11 display forwarding in SSH (or other secure protocols), combined with actual and pseudo-displays that accept connections only over local IPC mechanisms authorized by permissions or access control lists (ACLs), does correct many X11 security problems, as long as the "none" MAC is not used. It is RECOMMENDED that X11 display implementations default to allow the display to open only over local IPC. It is RECOMMENDED that SSH server implementations that support X11 forwarding default to allow the display to open only over local IPC. On single-user systems, it might be reasonable to default to allow the local display to open over TCP/IP.

Implementers of the X11 forwarding protocol SHOULD implement the magic cookie access-checking spoofing mechanism, as described in [SSH-CONNECT], as an additional mechanism to prevent unauthorized use of the proxy.

10. References

10.1. Normative References

[SSH-TRANS] Ylonen, T. and C. Lonvick, Ed., "The Secure Shell (SSH) Transport Layer Protocol", RFC 4253, January 2006.

[SSH-USERAUTH] Ylonen, T. and C. Lonvick, Ed., "The Secure Shell (SSH) Authentication Protocol", RFC 4252, January 2006.

[SSH-CONNECT] Ylonen, T. and C. Lonvick, Ed., "The Secure Shell (SSH) Connection Protocol", RFC 4254, January 2006.

[SSH-NUMBERS] Lehtinen, S. and C. Lonvick, Ed., "The Secure Shell (SSH) Protocol Assigned Numbers", RFC 4250, January 2006.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2434] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 2434, October 1998.

[RFC3066] Alvestrand, H., "Tags for the Identification of Languages", BCP 47, RFC 3066, January 2001.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, November 2003.

10.2. Informative References

[RFC0822] Crocker, D., "Standard for the format of ARPA Internet text messages", STD 11, RFC 822, August 1982.

[RFC0854] Postel, J. and J. Reynolds, "Telnet Protocol Specification", STD 8, RFC 854, May 1983.

[RFC1034] Mockapetris, P., "Domain names - concepts and facilities", STD 13, RFC 1034, November 1987.

[RFC1282] Kantor, B., "BSD Rlogin", RFC 1282, December 1991.

[RFC4120] Neuman, C., Yu, T., Hartman, S., and K. Raeburn, "The Kerberos Network Authentication Service (V5)", RFC 4120, July 2005.

[RFC1964] Linn, J., "The Kerberos Version 5 GSS-API Mechanism", RFC 1964, June 1996.

[RFC2025] Adams, C., "The Simple Public-Key GSS-API Mechanism (SPKM)", RFC 2025, October 1996.

[RFC2085] Oehler, M. and R. Glenn, "HMAC-MD5 IP Authentication with Replay Prevention", RFC 2085, February 1997.

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, February 1997.

[RFC2246] Dierks, T. and C. Allen, "The TLS Protocol Version 1.0", RFC 2246, January 1999.

[RFC2410] Glenn, R. and S. Kent, "The NULL Encryption Algorithm and Its Use With IPsec", RFC 2410, November 1998.

[RFC2743] Linn, J., "Generic Security Service Application Program Interface Version 2, Update 1", RFC 2743, January 2000.

[RFC3766] Orman, H. and P. Hoffman, "Determining Strengths For Public Keys Used For Exchanging Symmetric Keys", BCP 86, RFC 3766, April 2004.

[RFC4086] Eastlake, D., 3rd, Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, June 2005.

[FIPS-180-2] US National Institute of Standards and Technology, "Secure Hash Standard (SHS)", Federal Information Processing Standards Publication 180-2, August 2002.

[FIPS-186-2] US National Institute of Standards and Technology, "Digital Signature Standard (DSS)", Federal Information Processing Standards Publication 186-2, January 2000.

[FIPS-197] US National Institute of Standards and Technology, "Advanced Encryption Standard (AES)", Federal Information Processing Standards Publication 197, November 2001.

[ANSI-T1.523-2001] American National Standards Institute, Inc., "Telecom Glossary 2000", ANSI T1.523-2001, February 2001.

[SCHNEIER] Schneier, B., "Applied Cryptography Second Edition: protocols algorithms and source in code in C", John Wiley and Sons, New York, NY, 1996.

[SCHEIFLER] Scheifler, R., "X Window System : The Complete Reference to Xlib, X Protocol, Icccm, Xlfd, 3rd edition.", Digital Press, ISBN 1555580882, February 1992.

[KAUFMAN] Kaufman, C., Perlman, R., and M. Speciner, "Network Security: PRIVATE Communication in a PUBLIC World", Prentice Hall Publisher, 1995.

[CERT] CERT Coordination Center, The., "http://www.cert.org/nav/index_red.html".

[VENEMA] Venema, W., "Murphy's Law and Computer Security", Proceedings of 6th USENIX Security Symposium, San Jose CA http://www.usenix.org/publications/library/ proceedings/sec96/venema.html, July 1996.

[ROGAWAY] Rogaway, P., "Problems with Proposed IP Cryptography", Unpublished paper http://www.cs.ucdavis.edu/~rogaway/ papers/draft-rogaway-ipsec-comments-00.txt, 1996.

[DAI] Dai, W., "An attack against SSH2 protocol", Email to the SECSH Working Group ietf-ssh@netbsd.org ftp:// ftp.ietf.org/ietf-mail-archive/secsh/2002-02.mail, Feb 2002.

[BELLARE] Bellaire, M., Kohno, T., and C. Namprempre, "Authenticated Encryption in SSH: Fixing the SSH Binary Packet Protocol", Proceedings of the 9th ACM Conference on Computer and Communications Security, Sept 2002.

[Openwall] Solar Designer and D. Song, "SSH Traffic Analysis Attacks", Presentation given at HAL2001 and NordU2002 Conferences, Sept 2001.

[USENIX] Song, X.D., Wagner, D., and X. Tian, "Timing Analysis of Keystrokes and SSH Timing Attacks", Paper given at 10th USENIX Security Symposium, 2001.

Authors' Addresses

Tatu Ylonen
SSH Communications Security Corp
Valimotie 17
00380 Helsinki
Finland

EMail: ylo@ssh.com

Chris Lonvick (editor)
Cisco Systems, Inc.
12515 Research Blvd.
Austin 78759
USA

EMail: clonvick@cisco.com

Trademark Notice

Full Copyright Statement

Intellectual Property

Acknowledgement

Funding for the RFC Editor function is provided by the IETF Administrative Support Activity (IASA).

Exhibit 2

Network Working Group
Request for Comments: 4252
Category: Standards Track

T. Ylonen
SSH Communications Security Corp
C. Lonvick, Ed.
Cisco Systems, Inc.

January 2006

The Secure Shell (SSH) Authentication Protocol

Status of This Memo

Copyright Notice

Abstract

The Secure Shell Protocol (SSH) is a protocol for secure remote login and other secure network services over an insecure network. This document describes the SSH authentication protocol framework and public key, password, and host-based client authentication methods. Additional authentication methods are described in separate documents. The SSH authentication protocol runs on top of the SSH transport layer protocol and provides a single authenticated tunnel for the SSH connection protocol.

Table of Contents

1. Introduction

The SSH authentication protocol is a general-purpose user authentication protocol. It is intended to be run over the SSH transport layer protocol [SSH-TRANS]. This protocol assumes that the underlying protocols provide integrity and confidentiality protection.

This document should be read only after reading the SSH architecture document [SSH-ARCH]. This document freely uses terminology and notation from the architecture document without reference or further explanation.

The 'service name' for this protocol is "ssh-userauth".

When this protocol starts, it receives the session identifier from the lower-level protocol (this is the exchange hash H from the first key exchange). The session identifier uniquely identifies this session and is suitable for signing in order to prove ownership of a private key. This protocol also needs to know whether the lower-level protocol provides confidentiality protection.

2. Contributors

The major original contributors of this set of documents have been: Tatu Ylonen, Tero Kivinen, Timo J. Rinne, Sami Lehtinen (all of SSH Communications Security Corp), and Markku-Juhani O. Saarinen (University of Jyvaskyla). Darren Moffat was the original editor of this set of documents and also made very substantial contributions.

Many people contributed to the development of this document over the years. People who should be acknowledged include Mats Andersson, Ben Harris, Bill Sommerfeld, Brent McClure, Niels Moller, Damien Miller, Derek Fawcus, Frank Cusack, Heikki Nousiainen, Jakob Schlyter, Jeff Van Dyke, Jeffrey Altman, Jeffrey Hutzelman, Jon Bright, Joseph Galbraith, Ken Hornstein, Markus Friedl, Martin Forssen, Nicolas Williams, Niels Provos, Perry Metzger, Peter Gutmann, Simon Josefsson, Simon Tatham, Wei Dai, Denis Bider, der Mouse, and Tadayoshi Kohno. Listing their names here does not mean that they endorse this document, but that they have contributed to it.

3. Conventions Used in This Document

All documents related to the SSH protocols shall use the keywords "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" to describe requirements. These keywords are to be interpreted as described in [RFC2119].

The keywords "PRIVATE USE", "HIERARCHICAL ALLOCATION", "FIRST COME FIRST SERVED", "EXPERT REVIEW", "SPECIFICATION REQUIRED", "IESG APPROVAL", "IETF CONSENSUS", and "STANDARDS ACTION" that appear in this document when used to describe namespace allocation are to be interpreted as described in [RFC2434].

Protocol fields and possible values to fill them are defined in this set of documents. Protocol fields will be defined in the message definitions. As an example, SSH_MSG_CHANNEL_DATA is defined as follows.

```
byte      SSH_MSG_CHANNEL_DATA
uint32    recipient channel
string    data
```

Throughout these documents, when the fields are referenced, they will appear within single quotes. When values to fill those fields are referenced, they will appear within double quotes. Using the above example, possible values for ’data’ are "foo" and "bar".

4. The Authentication Protocol Framework

The server drives the authentication by telling the client which authentication methods can be used to continue the exchange at any given time. The client has the freedom to try the methods listed by the server in any order. This gives the server complete control over the authentication process if desired, but also gives enough flexibility for the client to use the methods it supports or that are most convenient for the user, when multiple methods are offered by the server.

Authentication methods are identified by their name, as defined in [SSH-ARCH]. The "none" method is reserved, and MUST NOT be listed as supported. However, it MAY be sent by the client. The server MUST always reject this request, unless the client is to be granted access without any authentication, in which case, the server MUST accept this request. The main purpose of sending this request is to get the list of supported methods from the server.

The server SHOULD have a timeout for authentication and disconnect if the authentication has not been accepted within the timeout period. The RECOMMENDED timeout period is 10 minutes. Additionally, the implementation SHOULD limit the number of failed authentication attempts a client may perform in a single session (the RECOMMENDED limit is 20 attempts). If the threshold is exceeded, the server SHOULD disconnect.

Additional thoughts about authentication timeouts and retries may be found in [ssh-1.2.30].

5. Authentication Requests

All authentication requests MUST use the following message format. Only the first few fields are defined; the remaining fields depend on the authentication method.

```
byte      SSH_MSG_USERAUTH_REQUEST
string    user name in ISO-10646 UTF-8 encoding [RFC3629]
string    service name in US-ASCII
string    method name in US-ASCII
....      method specific fields
```

The 'user name' and 'service name' are repeated in every new authentication attempt, and MAY change. The server implementation MUST carefully check them in every message, and MUST flush any accumulated authentication states if they change. If it is unable to

flush an authentication state, it MUST disconnect if the 'user name' or 'service name' changes.

The 'service name' specifies the service to start after authentication. There may be several different authenticated services provided. If the requested service is not available, the server MAY disconnect immediately or at any later time. Sending a proper disconnect message is RECOMMENDED. In any case, if the service does not exist, authentication MUST NOT be accepted.

If the requested 'user name' does not exist, the server MAY disconnect, or MAY send a bogus list of acceptable authentication 'method name' values, but never accept any. This makes it possible for the server to avoid disclosing information on which accounts exist. In any case, if the 'user name' does not exist, the authentication request MUST NOT be accepted.

While there is usually little point for clients to send requests that the server does not list as acceptable, sending such requests is not an error, and the server SHOULD simply reject requests that it does not recognize.

An authentication request MAY result in a further exchange of messages. All such messages depend on the authentication 'method name' used, and the client MAY at any time continue with a new SSH_MSG_USERAUTH_REQUEST message, in which case the server MUST abandon the previous authentication attempt and continue with the new one.

The following 'method name' values are defined.

"publickey"	REQUIRED
"password"	OPTIONAL
"hostbased"	OPTIONAL
"none"	NOT RECOMMENDED

Additional 'method name' values may be defined as specified in [SSH-ARCH] and [SSH-NUMBERS].

5.1. Responses to Authentication Requests

If the server rejects the authentication request, it MUST respond with the following:

byte	SSH_MSG_USERAUTH_FAILURE
name-list	authentications that can continue
boolean	partial success

The 'authentications that can continue' is a comma-separated name-list of authentication 'method name' values that may productively continue the authentication dialog.

It is RECOMMENDED that servers only include those 'method name' values in the name-list that are actually useful. However, it is not illegal to include 'method name' values that cannot be used to authenticate the user.

Already successfully completed authentications SHOULD NOT be included in the name-list, unless they should be performed again for some reason.

The value of 'partial success' MUST be TRUE if the authentication request to which this is a response was successful. It MUST be FALSE if the request was not successfully processed.

When the server accepts authentication, it MUST respond with the following:

```
byte      SSH_MSG_USERAUTH_SUCCESS
```

Note that this is not sent after each step in a multi-method authentication sequence, but only when the authentication is complete.

The client MAY send several authentication requests without waiting for responses from previous requests. The server MUST process each request completely and acknowledge any failed requests with a SSH_MSG_USERAUTH_FAILURE message before processing the next request.

A request that requires further messages to be exchanged will be aborted by a subsequent request. A client MUST NOT send a subsequent request if it has not received a response from the server for a previous request. A SSH_MSG_USERAUTH_FAILURE message MUST NOT be sent for an aborted method.

SSH_MSG_USERAUTH_SUCCESS MUST be sent only once. When SSH_MSG_USERAUTH_SUCCESS has been sent, any further authentication requests received after that SHOULD be silently ignored.

Any non-authentication messages sent by the client after the request that resulted in SSH_MSG_USERAUTH_SUCCESS being sent MUST be passed to the service being run on top of this protocol. Such messages can be identified by their message numbers (see Section 6).

5.2. The "none" Authentication Request

A client may request a list of authentication 'method name' values that may continue by using the "none" authentication 'method name'.

If no authentication is needed for the user, the server MUST return SSH_MSG_USERAUTH_SUCCESS. Otherwise, the server MUST return SSH_MSG_USERAUTH_FAILURE and MAY return with it a list of methods that may continue in its 'authentications that can continue' value.

This 'method name' MUST NOT be listed as supported by the server.

5.3. Completion of User Authentication

Authentication is complete when the server has responded with SSH_MSG_USERAUTH_SUCCESS. All authentication related messages received after sending this message SHOULD be silently ignored.

After sending SSH_MSG_USERAUTH_SUCCESS, the server starts the requested service.

5.4. Banner Message

In some jurisdictions, sending a warning message before authentication may be relevant for getting legal protection. Many UNIX machines, for example, normally display text from /etc/issue, use TCP wrappers, or similar software to display a banner before issuing a login prompt.

The SSH server may send an SSH_MSG_USERAUTH_BANNER message at any time after this authentication protocol starts and before authentication is successful. This message contains text to be displayed to the client user before authentication is attempted. The format is as follows:

```
byte      SSH_MSG_USERAUTH_BANNER
string    message in ISO-10646 UTF-8 encoding [RFC3629]
string    language tag [RFC3066]
```

By default, the client SHOULD display the 'message' on the screen. However, since the 'message' is likely to be sent for every login attempt, and since some client software will need to open a separate window for this warning, the client software may allow the user to explicitly disable the display of banners from the server. The 'message' may consist of multiple lines, with line breaks indicated by CRLF pairs.

If the 'message' string is displayed, control character filtering, discussed in [SSH-ARCH], SHOULD be used to avoid attacks by sending terminal control characters.

6. Authentication Protocol Message Numbers

All message numbers used by this authentication protocol are in the range from 50 to 79, which is part of the range reserved for protocols running on top of the SSH transport layer protocol.

Message numbers of 80 and higher are reserved for protocols running after this authentication protocol, so receiving one of them before authentication is complete is an error, to which the server MUST respond by disconnecting, preferably with a proper disconnect message sent to ease troubleshooting.

After successful authentication, such messages are passed to the higher-level service.

These are the general authentication message codes:

SSH_MSG_USERAUTH_REQUEST	50
SSH_MSG_USERAUTH_FAILURE	51
SSH_MSG_USERAUTH_SUCCESS	52
SSH_MSG_USERAUTH_BANNER	53

In addition to the above, there is a range of message numbers (60 to 79) reserved for method-specific messages. These messages are only sent by the server (client sends only SSH_MSG_USERAUTH_REQUEST messages). Different authentication methods reuse the same message numbers.

7. Public Key Authentication Method: "publickey"

The only REQUIRED authentication 'method name' is "publickey" authentication. All implementations MUST support this method; however, not all users need to have public keys, and most local policies are not likely to require public key authentication for all users in the near future.

With this method, the possession of a private key serves as authentication. This method works by sending a signature created with a private key of the user. The server MUST check that the key is a valid authenticator for the user, and MUST check that the signature is valid. If both hold, the authentication request MUST be accepted; otherwise, it MUST be rejected. Note that the server MAY require additional authentications after successful authentication.

Private keys are often stored in an encrypted form at the client host, and the user must supply a passphrase before the signature can be generated. Even if they are not, the signing operation involves some expensive computation. To avoid unnecessary processing and user interaction, the following message is provided for querying whether authentication using the "publickey" method would be acceptable.

```
byte      SSH_MSG_USERAUTH_REQUEST
string    user name in ISO-10646 UTF-8 encoding [RFC3629]
string    service name in US-ASCII
string    "publickey"
boolean   FALSE
string    public key algorithm name
string    public key blob
```

Public key algorithms are defined in the transport layer specification [SSH-TRANS]. The 'public key blob' may contain certificates.

Any public key algorithm may be offered for use in authentication. In particular, the list is not constrained by what was negotiated during key exchange. If the server does not support some algorithm, it MUST simply reject the request.

The server MUST respond to this message with either SSH_MSG_USERAUTH_FAILURE or with the following:

```
byte      SSH_MSG_USERAUTH_PK_OK
string    public key algorithm name from the request
string    public key blob from the request
```

To perform actual authentication, the client MAY then send a signature generated using the private key. The client MAY send the signature directly without first verifying whether the key is acceptable. The signature is sent using the following packet:

```
byte      SSH_MSG_USERAUTH_REQUEST
string    user name
string    service name
string    "publickey"
boolean   TRUE
string    public key algorithm name
string    public key to be used for authentication
string    signature
```

The value of 'signature' is a signature by the corresponding private key over the following data, in the following order:

```
string    session identifier
byte      SSH_MSG_USERAUTH_REQUEST
string    user name
string    service name
string    "publickey"
boolean   TRUE
string    public key algorithm name
string    public key to be used for authentication
```

When the server receives this message, it MUST check whether the supplied key is acceptable for authentication, and if so, it MUST check whether the signature is correct.

If both checks succeed, this method is successful. Note that the server may require additional authentications. The server MUST respond with SSH_MSG_USERAUTH_SUCCESS (if no more authentications are needed), or SSH_MSG_USERAUTH_FAILURE (if the request failed, or more authentications are needed).

The following method-specific message numbers are used by the "publickey" authentication method.

```
SSH_MSG_USERAUTH_PK_OK              60
```

8. Password Authentication Method: "password"

Password authentication uses the following packets. Note that a server MAY request that a user change the password. All implementations SHOULD support password authentication.

```
byte      SSH_MSG_USERAUTH_REQUEST
string    user name
string    service name
string    "password"
boolean   FALSE
string    plaintext password in ISO-10646 UTF-8 encoding [RFC3629]
```

Note that the 'plaintext password' value is encoded in ISO-10646 UTF-8. It is up to the server how to interpret the password and validate it against the password database. However, if the client reads the password in some other encoding (e.g., ISO 8859-1 - ISO Latin1), it MUST convert the password to ISO-10646 UTF-8 before transmitting, and the server MUST convert the password to the encoding used on that system for passwords.

From an internationalization standpoint, it is desired that if a user enters their password, the authentication process will work regardless of what OS and client software the user is using. Doing so requires normalization. Systems supporting non-ASCII passwords SHOULD always normalize passwords and user names whenever they are added to the database, or compared (with or without hashing) to existing entries in the database. SSH implementations that both store the passwords and compare them SHOULD use [RFC4013] for normalization.

Note that even though the cleartext password is transmitted in the packet, the entire packet is encrypted by the transport layer. Both the server and the client should check whether the underlying transport layer provides confidentiality (i.e., if encryption is being used). If no confidentiality is provided ("none" cipher), password authentication SHOULD be disabled. If there is no confidentiality or no MAC, password change SHOULD be disabled.

Normally, the server responds to this message with success or failure. However, if the password has expired, the server SHOULD indicate this by responding with SSH_MSG_USERAUTH_PASSWD_CHANGEREQ. In any case, the server MUST NOT allow an expired password to be used for authentication.

```
byte      SSH_MSG_USERAUTH_PASSWD_CHANGEREQ
string    prompt in ISO-10646 UTF-8 encoding [RFC3629]
string    language tag [RFC3066]
```

In this case, the client MAY continue with a different authentication method, or request a new password from the user and retry password authentication using the following message. The client MAY also send this message instead of the normal password authentication request without the server asking for it.

```
byte      SSH_MSG_USERAUTH_REQUEST
string    user name
string    service name
string    "password"
boolean   TRUE
string    plaintext old password in ISO-10646 UTF-8 encoding
           [RFC3629]
string    plaintext new password in ISO-10646 UTF-8 encoding
           [RFC3629]
```

The server must reply to each request message with SSH_MSG_USERAUTH_SUCCESS, SSH_MSG_USERAUTH_FAILURE, or another SSH_MSG_USERAUTH_PASSWD_CHANGEREQ. The meaning of these is as follows:

SSH_MSG_USERAUTH_SUCCESS - The password has been changed, and authentication has been successfully completed.

SSH_MSG_USERAUTH_FAILURE with partial success - The password has been changed, but more authentications are needed.

SSH_MSG_USERAUTH_FAILURE without partial success - The password has not been changed. Either password changing was not supported, or the old password was bad. Note that if the server has already sent SSH_MSG_USERAUTH_PASSWD_CHANGEREQ, we know that it supports changing the password.

SSH_MSG_USERAUTH_CHANGEREQ - The password was not changed because the new password was not acceptable (e.g., too easy to guess).

The following method-specific message numbers are used by the password authentication method.

SSH_MSG_USERAUTH_PASSWD_CHANGEREQ 60

9. Host-Based Authentication: "hostbased"

Some sites wish to allow authentication based on the host that the user is coming from and the user name on the remote host. While this form of authentication is not suitable for high-security sites, it can be very convenient in many environments. This form of authentication is OPTIONAL. When used, special care SHOULD be taken to prevent a regular user from obtaining the private host key.

The client requests this form of authentication by sending the following message. It is similar to the UNIX "rhosts" and "hosts.equiv" styles of authentication, except that the identity of the client host is checked more rigorously.

This method works by having the client send a signature created with the private key of the client host, which the server checks with that host's public key. Once the client host's identity is established, authorization (but no further authentication) is performed based on the user names on the server and the client, and the client host name.

```
byte      SSH_MSG_USERAUTH_REQUEST
string    user name
string    service name
string    "hostbased"
string    public key algorithm for host key
string    public host key and certificates for client host
string    client host name expressed as the FQDN in US-ASCII
string    user name on the client host in ISO-10646 UTF-8 encoding
           [RFC3629]
string    signature
```

Public key algorithm names for use in 'public key algorithm for host key' are defined in the transport layer specification [SSH-TRANS]. The 'public host key and certificates for client host' may include certificates.

The value of 'signature' is a signature with the private host key of the following data, in this order:

```
string    session identifier
byte      SSH_MSG_USERAUTH_REQUEST
string    user name
string    service name
string    "hostbased"
string    public key algorithm for host key
string    public host key and certificates for client host
string    client host name expressed as the FQDN in US-ASCII
string    user name on the client host in ISO-10646 UTF-8 encoding
           [RFC3629]
```

The server MUST verify that the host key actually belongs to the client host named in the message, that the given user on that host is allowed to log in, and that the 'signature' value is a valid signature on the appropriate value by the given host key. The server MAY ignore the client 'user name', if it wants to authenticate only the client host.

Whenever possible, it is RECOMMENDED that the server perform additional checks to verify that the network address obtained from the (untrusted) network matches the given client host name. This makes exploiting compromised host keys more difficult. Note that this may require special handling for connections coming through a firewall.

10. IANA Considerations

This document is part of a set. The IANA considerations for the SSH protocol, as defined in [SSH-ARCH], [SSH-TRANS], [SSH-CONNECT], and this document, are detailed in [SSH-NUMBERS].

11. Security Considerations

The purpose of this protocol is to perform client user authentication. It assumed that this runs over a secure transport layer protocol, which has already authenticated the server machine, established an encrypted communications channel, and computed a unique session identifier for this session. The transport layer provides forward secrecy for password authentication and other methods that rely on secret data.

Full security considerations for this protocol are provided in [SSH-ARCH].

12. References

12.1. Normative References

[SSH-ARCH] Ylonen, T. and C. Lonvick, Ed., "The Secure Shell (SSH) Protocol Architecture", RFC 4251, January 2006.

[SSH-CONNECT] Ylonen, T. and C. Lonvick, Ed., "The Secure Shell (SSH) Connection Protocol", RFC 4254, January 2006.

[SSH-TRANS] Ylonen, T. and C. Lonvick, Ed., "The Secure Shell (SSH) Transport Layer Protocol", RFC 4253, January 2006.

[SSH-NUMBERS] Lehtinen, S. and C. Lonvick, Ed., "The Secure Shell (SSH) Protocol Assigned Numbers", RFC 4250, January 2006.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2434] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 2434, October 1998.

[RFC3066] Alvestrand, H., "Tags for the Identification of Languages", BCP 47, RFC 3066, January 2001.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, November 2003.

[RFC4013] Zeilenga, K., "SASLprep: Stringprep Profile for User Names and Passwords", RFC 4013, February 2005.

12.2. Informative References

[ssh-1.2.30] Ylonen, T., "ssh-1.2.30/RFC", File within compressed tarball ftp://ftp.funet.fi/pub/unix/security/login/ssh/ssh-1.2.30.tar.gz, November 1995.

Authors' Addresses

Tatu Ylonen
SSH Communications Security Corp
Valimotie 17
00380 Helsinki
Finland

EMail: ylo@ssh.com

Chris Lonvick (editor)
Cisco Systems, Inc.
12515 Research Blvd.
Austin 78759
USA

EMail: clonvick@cisco.com

Trademark Notice

"ssh" is a registered trademark in the United States and/or other countries.

Full Copyright Statement

Intellectual Property

Acknowledgement

Funding for the RFC Editor function is provided by the IETF Administrative Support Activity (IASA).

Exhibit 3

Network Working Group
Request for Comments: 4253
Category: Standards Track

T. Ylonen
SSH Communications Security Corp
C. Lonvick, Ed.
Cisco Systems, Inc.

January 2006

The Secure Shell (SSH) Transport Layer Protocol

Status of This Memo

Copyright Notice

Abstract

The Secure Shell (SSH) is a protocol for secure remote login and other secure network services over an insecure network.

This document describes the SSH transport layer protocol, which typically runs on top of TCP/IP. The protocol can be used as a basis for a number of secure network services. It provides strong encryption, server authentication, and integrity protection. It may also provide compression.

Key exchange method, public key algorithm, symmetric encryption algorithm, message authentication algorithm, and hash algorithm are all negotiated.

This document also describes the Diffie-Hellman key exchange method and the minimal set of algorithms that are needed to implement the SSH transport layer protocol.

Table of Contents

1. Introduction

The SSH transport layer is a secure, low level transport protocol. It provides strong encryption, cryptographic host authentication, and integrity protection.

Authentication in this protocol level is host-based; this protocol does not perform user authentication. A higher level protocol for user authentication can be designed on top of this protocol.

The protocol has been designed to be simple and flexible to allow parameter negotiation, and to minimize the number of round-trips. The key exchange method, public key algorithm, symmetric encryption algorithm, message authentication algorithm, and hash algorithm are all negotiated. It is expected that in most environments, only 2 round-trips will be needed for full key exchange, server authentication, service request, and acceptance notification of service request. The worst case is 3 round-trips.

2. Contributors

The major original contributors of this set of documents have been: Tatu Ylonen, Tero Kivinen, Timo J. Rinne, Sami Lehtinen (all of SSH Communications Security Corp), and Markku-Juhani O. Saarinen (University of Jyvaskyla). Darren Moffat was the original editor of this set of documents and also made very substantial contributions.

Many people contributed to the development of this document over the years. People who should be acknowledged include Mats Andersson, Ben Harris, Bill Sommerfeld, Brent McClure, Niels Moller, Damien Miller, Derek Fawcus, Frank Cusack, Heikki Nousiainen, Jakob Schlyter, Jeff Van Dyke, Jeffrey Altman, Jeffrey Hutzelman, Jon Bright, Joseph Galbraith, Ken Hornstein, Markus Friedl, Martin Forssen, Nicolas Williams, Niels Provos, Perry Metzger, Peter Gutmann, Simon Josefsson, Simon Tatham, Wei Dai, Denis Bider, der Mouse, and Tadayoshi Kohno. Listing their names here does not mean that they endorse this document, but that they have contributed to it.

3. Conventions Used in This Document

All documents related to the SSH protocols shall use the keywords "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" to describe requirements. These keywords are to be interpreted as described in [RFC2119].

The keywords "PRIVATE USE", "HIERARCHICAL ALLOCATION", "FIRST COME FIRST SERVED", "EXPERT REVIEW", "SPECIFICATION REQUIRED", "IESG APPROVAL", "IETF CONSENSUS", and "STANDARDS ACTION" that appear in this document when used to describe namespace allocation are to be interpreted as described in [RFC2434].

Protocol fields and possible values to fill them are defined in this set of documents. Protocol fields will be defined in the message definitions. As an example, SSH_MSG_CHANNEL_DATA is defined as follows.

```
byte      SSH_MSG_CHANNEL_DATA
uint32    recipient channel
string    data
```

Throughout these documents, when the fields are referenced, they will appear within single quotes. When values to fill those fields are referenced, they will appear within double quotes. Using the above example, possible values for 'data' are "foo" and "bar".

4. Connection Setup

SSH works over any 8-bit clean, binary-transparent transport. The underlying transport SHOULD protect against transmission errors, as such errors cause the SSH connection to terminate.

The client initiates the connection.

4.1. Use over TCP/IP

When used over TCP/IP, the server normally listens for connections on port 22. This port number has been registered with the IANA, and has been officially assigned for SSH.

4.2. Protocol Version Exchange

When the connection has been established, both sides MUST send an identification string. This identification string MUST be

```
SSH-protoversion-softwareversion SP comments CR LF
```

Since the protocol being defined in this set of documents is version 2.0, the 'protoversion' MUST be "2.0". The 'comments' string is OPTIONAL. If the 'comments' string is included, a 'space' character (denoted above as SP, ASCII 32) MUST separate the 'softwareversion' and 'comments' strings. The identification MUST be terminated by a single Carriage Return (CR) and a single Line Feed (LF) character (ASCII 13 and 10, respectively). Implementers who wish to maintain

compatibility with older, undocumented versions of this protocol may want to process the identification string without expecting the presence of the carriage return character for reasons described in Section 5 of this document. The null character MUST NOT be sent. The maximum length of the string is 255 characters, including the Carriage Return and Line Feed.

The part of the identification string preceding the Carriage Return and Line Feed is used in the Diffie-Hellman key exchange (see Section 8).

The server MAY send other lines of data before sending the version string. Each line SHOULD be terminated by a Carriage Return and Line Feed. Such lines MUST NOT begin with "SSH-", and SHOULD be encoded in ISO-10646 UTF-8 [RFC3629] (language is not specified). Clients MUST be able to process such lines. Such lines MAY be silently ignored, or MAY be displayed to the client user. If they are displayed, control character filtering, as discussed in [SSH-ARCH], SHOULD be used. The primary use of this feature is to allow TCP-wrappers to display an error message before disconnecting.

Both the 'protoversion' and 'softwareversion' strings MUST consist of printable US-ASCII characters, with the exception of whitespace characters and the minus sign (-). The 'softwareversion' string is primarily used to trigger compatibility extensions and to indicate the capabilities of an implementation. The 'comments' string SHOULD contain additional information that might be useful in solving user problems. As such, an example of a valid identification string is

```
SSH-2.0-billsSSH_3.6.3q3<CR><LF>
```

This identification string does not contain the optional 'comments' string and is thus terminated by a CR and LF immediately after the 'softwareversion' string.

Key exchange will begin immediately after sending this identifier. All packets following the identification string SHALL use the binary packet protocol, which is described in Section 6.

5. Compatibility With Old SSH Versions

As stated earlier, the 'protoversion' specified for this protocol is "2.0". Earlier versions of this protocol have not been formally documented, but it is widely known that they use 'protoversion' of "1.x" (e.g., "1.5" or "1.3"). At the time of this writing, many implementations of SSH are utilizing protocol version 2.0, but it is known that there are still devices using the previous versions. During the transition period, it is important to be able to work in a

way that is compatible with the installed SSH clients and servers that use the older version of the protocol. Information in this section is only relevant for implementations supporting compatibility with SSH versions 1.x. For those interested, the only known documentation of the 1.x protocol is contained in README files that are shipped along with the source code [ssh-1.2.30].

5.1. Old Client, New Server

Server implementations MAY support a configurable compatibility flag that enables compatibility with old versions. When this flag is on, the server SHOULD identify its 'protoversion' as "1.99". Clients using protocol 2.0 MUST be able to identify this as identical to "2.0". In this mode, the server SHOULD NOT send the Carriage Return character (ASCII 13) after the identification string.

In the compatibility mode, the server SHOULD NOT send any further data after sending its identification string until it has received an identification string from the client. The server can then determine whether the client is using an old protocol, and can revert to the old protocol if required. In the compatibility mode, the server MUST NOT send additional data before the identification string.

When compatibility with old clients is not needed, the server MAY send its initial key exchange data immediately after the identification string.

5.2. New Client, Old Server

Since the new client MAY immediately send additional data after its identification string (before receiving the server's identification string), the old protocol may already be corrupt when the client learns that the server is old. When this happens, the client SHOULD close the connection to the server, and reconnect using the old protocol.

5.3. Packet Size and Overhead

Some readers will worry about the increase in packet size due to new headers, padding, and the Message Authentication Code (MAC). The minimum packet size is in the order of 28 bytes (depending on negotiated algorithms). The increase is negligible for large packets, but very significant for one-byte packets (telnet-type sessions). There are, however, several factors that make this a non-issue in almost all cases:

- The minimum size of a TCP/IP header is 32 bytes. Thus, the increase is actually from 33 to 51 bytes (roughly).

- The minimum size of the data field of an Ethernet packet is 46 bytes [RFC0894]. Thus, the increase is no more than 5 bytes. When Ethernet headers are considered, the increase is less than 10 percent.

- The total fraction of telnet-type data in the Internet is negligible, even with increased packet sizes.

The only environment where the packet size increase is likely to have a significant effect is PPP [RFC1661] over slow modem lines (PPP compresses the TCP/IP headers, emphasizing the increase in packet size). However, with modern modems, the time needed to transfer is in the order of 2 milliseconds, which is a lot faster than people can type.

There are also issues related to the maximum packet size. To minimize delays in screen updates, one does not want excessively large packets for interactive sessions. The maximum packet size is negotiated separately for each channel.

6. Binary Packet Protocol

Each packet is in the following format:

```
uint32    packet_length
byte      padding_length
byte[n1]  payload; n1 = packet_length - padding_length - 1
byte[n2]  random padding; n2 = padding_length
byte[m]   mac (Message Authentication Code - MAC); m = mac_length
```

packet_length
: The length of the packet in bytes, not including 'mac' or the 'packet_length' field itself.

padding_length
: Length of 'random padding' (bytes).

payload
: The useful contents of the packet. If compression has been negotiated, this field is compressed. Initially, compression MUST be "none".

random padding
: Arbitrary-length padding, such that the total length of (packet_length || padding_length || payload || random padding) is a multiple of the cipher block size or 8, whichever is

larger. There MUST be at least four bytes of padding. The padding SHOULD consist of random bytes. The maximum amount of padding is 255 bytes.

mac
Message Authentication Code. If message authentication has been negotiated, this field contains the MAC bytes. Initially, the MAC algorithm MUST be "none".

Note that the length of the concatenation of 'packet_length', 'padding_length', 'payload', and 'random padding' MUST be a multiple of the cipher block size or 8, whichever is larger. This constraint MUST be enforced, even when using stream ciphers. Note that the 'packet_length' field is also encrypted, and processing it requires special care when sending or receiving packets. Also note that the insertion of variable amounts of 'random padding' may help thwart traffic analysis.

The minimum size of a packet is 16 (or the cipher block size, whichever is larger) bytes (plus 'mac'). Implementations SHOULD decrypt the length after receiving the first 8 (or cipher block size, whichever is larger) bytes of a packet.

6.1. Maximum Packet Length

All implementations MUST be able to process packets with an uncompressed payload length of 32768 bytes or less and a total packet size of 35000 bytes or less (including 'packet_length', 'padding_length', 'payload', 'random padding', and 'mac'). The maximum of 35000 bytes is an arbitrarily chosen value that is larger than the uncompressed length noted above. Implementations SHOULD support longer packets, where they might be needed. For example, if an implementation wants to send a very large number of certificates, the larger packets MAY be sent if the identification string indicates that the other party is able to process them. However, implementations SHOULD check that the packet length is reasonable in order for the implementation to avoid denial of service and/or buffer overflow attacks.

6.2. Compression

If compression has been negotiated, the 'payload' field (and only it) will be compressed using the negotiated algorithm. The 'packet_length' field and 'mac' will be computed from the compressed payload. Encryption will be done after compression.

Compression MAY be stateful, depending on the method. Compression MUST be independent for each direction, and implementations MUST allow independent choosing of the algorithm for each direction. In practice however, it is RECOMMENDED that the compression method be the same in both directions.

The following compression methods are currently defined:

none	REQUIRED	no compression
zlib	OPTIONAL	ZLIB (LZ77) compression

The "zlib" compression is described in [RFC1950] and in [RFC1951]. The compression context is initialized after each key exchange, and is passed from one packet to the next, with only a partial flush being performed at the end of each packet. A partial flush means that the current compressed block is ended and all data will be output. If the current block is not a stored block, one or more empty blocks are added after the current block to ensure that there are at least 8 bits, counting from the start of the end-of-block code of the current block to the end of the packet payload.

Additional methods may be defined as specified in [SSH-ARCH] and [SSH-NUMBERS].

6.3. Encryption

An encryption algorithm and a key will be negotiated during the key exchange. When encryption is in effect, the packet length, padding length, payload, and padding fields of each packet MUST be encrypted with the given algorithm.

The encrypted data in all packets sent in one direction SHOULD be considered a single data stream. For example, initialization vectors SHOULD be passed from the end of one packet to the beginning of the next packet. All ciphers SHOULD use keys with an effective key length of 128 bits or more.

The ciphers in each direction MUST run independently of each other. Implementations MUST allow the algorithm for each direction to be independently selected, if multiple algorithms are allowed by local policy. In practice however, it is RECOMMENDED that the same algorithm be used in both directions.

The following ciphers are currently defined:

3des-cbc	REQUIRED	three-key 3DES in CBC mode
blowfish-cbc	OPTIONAL	Blowfish in CBC mode
twofish256-cbc	OPTIONAL	Twofish in CBC mode, with a 256-bit key
twofish-cbc	OPTIONAL	alias for "twofish256-cbc" (this is being retained for historical reasons)
twofish192-cbc	OPTIONAL	Twofish with a 192-bit key
twofish128-cbc	OPTIONAL	Twofish with a 128-bit key
aes256-cbc	OPTIONAL	AES in CBC mode, with a 256-bit key
aes192-cbc	OPTIONAL	AES with a 192-bit key
aes128-cbc	RECOMMENDED	AES with a 128-bit key
serpent256-cbc	OPTIONAL	Serpent in CBC mode, with a 256-bit key
serpent192-cbc	OPTIONAL	Serpent with a 192-bit key
serpent128-cbc	OPTIONAL	Serpent with a 128-bit key
arcfour	OPTIONAL	the ARCFOUR stream cipher with a 128-bit key
idea-cbc	OPTIONAL	IDEA in CBC mode
cast128-cbc	OPTIONAL	CAST-128 in CBC mode
none	OPTIONAL	no encryption; NOT RECOMMENDED

The "3des-cbc" cipher is three-key triple-DES (encrypt-decrypt-encrypt), where the first 8 bytes of the key are used for the first encryption, the next 8 bytes for the decryption, and the following 8 bytes for the final encryption. This requires 24 bytes of key data (of which 168 bits are actually used). To implement CBC mode, outer chaining MUST be used (i.e., there is only one initialization vector). This is a block cipher with 8-byte blocks. This algorithm is defined in [FIPS-46-3]. Note that since this algorithm only has an effective key length of 112 bits ([SCHNEIER]), it does not meet the specifications that SSH encryption algorithms should use keys of 128 bits or more. However, this algorithm is still REQUIRED for historical reasons; essentially, all known implementations at the time of this writing support this algorithm, and it is commonly used because it is the fundamental interoperable algorithm. At some future time, it is expected that another algorithm, one with better strength, will become so prevalent and ubiquitous that the use of "3des-cbc" will be deprecated by another STANDARDS ACTION.

The "blowfish-cbc" cipher is Blowfish in CBC mode, with 128-bit keys [SCHNEIER]. This is a block cipher with 8-byte blocks.

The "twofish-cbc" or "twofish256-cbc" cipher is Twofish in CBC mode, with 256-bit keys as described [TWOFISH]. This is a block cipher with 16-byte blocks.

The "twofish192-cbc" cipher is the same as above, but with a 192-bit key.

The "twofish128-cbc" cipher is the same as above, but with a 128-bit key.

The "aes256-cbc" cipher is AES (Advanced Encryption Standard) [FIPS-197], in CBC mode. This version uses a 256-bit key.

The "aes192-cbc" cipher is the same as above, but with a 192-bit key.

The "aes128-cbc" cipher is the same as above, but with a 128-bit key.

The "serpent256-cbc" cipher in CBC mode, with a 256-bit key as described in the Serpent AES submission.

The "serpent192-cbc" cipher is the same as above, but with a 192-bit key.

The "serpent128-cbc" cipher is the same as above, but with a 128-bit key.

The "arcfour" cipher is the Arcfour stream cipher with 128-bit keys. The Arcfour cipher is believed to be compatible with the RC4 cipher [SCHNEIER]. Arcfour (and RC4) has problems with weak keys, and should be used with caution.

The "idea-cbc" cipher is the IDEA cipher in CBC mode [SCHNEIER].

The "cast128-cbc" cipher is the CAST-128 cipher in CBC mode with a 128-bit key [RFC2144].

The "none" algorithm specifies that no encryption is to be done. Note that this method provides no confidentiality protection, and it is NOT RECOMMENDED. Some functionality (e.g., password authentication) may be disabled for security reasons if this cipher is chosen.

Additional methods may be defined as specified in [SSH-ARCH] and in [SSH-NUMBERS].

6.4. Data Integrity

Data integrity is protected by including with each packet a MAC that is computed from a shared secret, packet sequence number, and the contents of the packet.

The message authentication algorithm and key are negotiated during key exchange. Initially, no MAC will be in effect, and its length MUST be zero. After key exchange, the 'mac' for the selected MAC algorithm will be computed before encryption from the concatenation of packet data:

```
mac = MAC(key, sequence_number || unencrypted_packet)
```

where unencrypted_packet is the entire packet without 'mac' (the length fields, 'payload' and 'random padding'), and sequence_number is an implicit packet sequence number represented as uint32. The sequence_number is initialized to zero for the first packet, and is incremented after every packet (regardless of whether encryption or MAC is in use). It is never reset, even if keys/algorithms are renegotiated later. It wraps around to zero after every 2^32 packets. The packet sequence_number itself is not included in the packet sent over the wire.

The MAC algorithms for each direction MUST run independently, and implementations MUST allow choosing the algorithm independently for both directions. In practice however, it is RECOMMENDED that the same algorithm be used in both directions.

The value of 'mac' resulting from the MAC algorithm MUST be transmitted without encryption as the last part of the packet. The number of 'mac' bytes depends on the algorithm chosen.

The following MAC algorithms are currently defined:

hmac-sha1	REQUIRED	HMAC-SHA1 (digest length = key length = 20)
hmac-sha1-96	RECOMMENDED	first 96 bits of HMAC-SHA1 (digest length = 12, key length = 20)
hmac-md5	OPTIONAL	HMAC-MD5 (digest length = key length = 16)
hmac-md5-96	OPTIONAL	first 96 bits of HMAC-MD5 (digest length = 12, key length = 16)
none	OPTIONAL	no MAC; NOT RECOMMENDED

The "hmac-*" algorithms are described in [RFC2104]. The "*-n" MACs use only the first n bits of the resulting value.

SHA-1 is described in [FIPS-180-2] and MD5 is described in [RFC1321].

Additional methods may be defined, as specified in [SSH-ARCH] and in [SSH-NUMBERS].

6.5. Key Exchange Methods

The key exchange method specifies how one-time session keys are generated for encryption and for authentication, and how the server authentication is done.

Two REQUIRED key exchange methods have been defined:

```
diffie-hellman-group1-sha1 REQUIRED
diffie-hellman-group14-sha1 REQUIRED
```

These methods are described in Section 8.

Additional methods may be defined as specified in [SSH-NUMBERS]. The name "diffie-hellman-group1-sha1" is used for a key exchange method using an Oakley group, as defined in [RFC2409]. SSH maintains its own group identifier space that is logically distinct from Oakley [RFC2412] and IKE; however, for one additional group, the Working Group adopted the number assigned by [RFC3526], using diffie-hellman-group14-sha1 for the name of the second defined group. Implementations should treat these names as opaque identifiers and should not assume any relationship between the groups used by SSH and the groups defined for IKE.

6.6. Public Key Algorithms

This protocol has been designed to operate with almost any public key format, encoding, and algorithm (signature and/or encryption).

There are several aspects that define a public key type:

- Key format: how is the key encoded and how are certificates represented. The key blobs in this protocol MAY contain certificates in addition to keys.

- Signature and/or encryption algorithms. Some key types may not support both signing and encryption. Key usage may also be restricted by policy statements (e.g., in certificates). In this case, different key types SHOULD be defined for the different policy alternatives.

- Encoding of signatures and/or encrypted data. This includes but is not limited to padding, byte order, and data formats.

The following public key and/or certificate formats are currently defined:

ssh-dss	REQUIRED	sign	Raw DSS Key
ssh-rsa	RECOMMENDED	sign	Raw RSA Key
pgp-sign-rsa	OPTIONAL	sign	OpenPGP certificates (RSA key)
pgp-sign-dss	OPTIONAL	sign	OpenPGP certificates (DSS key)

Additional key types may be defined, as specified in [SSH-ARCH] and in [SSH-NUMBERS].

The key type MUST always be explicitly known (from algorithm negotiation or some other source). It is not normally included in the key blob.

Certificates and public keys are encoded as follows:

string	certificate or public key format identifier
byte[n]	key/certificate data

The certificate part may be a zero length string, but a public key is required. This is the public key that will be used for authentication. The certificate sequence contained in the certificate blob can be used to provide authorization.

Public key/certificate formats that do not explicitly specify a signature format identifier MUST use the public key/certificate format identifier as the signature identifier.

Signatures are encoded as follows:

string	signature format identifier (as specified by the public key/certificate format)
byte[n]	signature blob in format specific encoding.

The "ssh-dss" key format has the following specific encoding:

string	"ssh-dss"
mpint	p
mpint	q
mpint	g
mpint	y

Here, the 'p', 'q', 'g', and 'y' parameters form the signature key blob.

Signing and verifying using this key format is done according to the Digital Signature Standard [FIPS-186-2] using the SHA-1 hash [FIPS-180-2].

The resulting signature is encoded as follows:

```
string    "ssh-dss"
string    dss_signature_blob
```

The value for 'dss_signature_blob' is encoded as a string containing r, followed by s (which are 160-bit integers, without lengths or padding, unsigned, and in network byte order).

The "ssh-rsa" key format has the following specific encoding:

```
string    "ssh-rsa"
mpint     e
mpint     n
```

Here the 'e' and 'n' parameters form the signature key blob.

Signing and verifying using this key format is performed according to the RSASSA-PKCS1-v1_5 scheme in [RFC3447] using the SHA-1 hash.

The resulting signature is encoded as follows:

```
string    "ssh-rsa"
string    rsa_signature_blob
```

The value for 'rsa_signature_blob' is encoded as a string containing s (which is an integer, without lengths or padding, unsigned, and in network byte order).

The "pgp-sign-rsa" method indicates the certificates, the public key, and the signature are in OpenPGP compatible binary format ([RFC2440]). This method indicates that the key is an RSA-key.

The "pgp-sign-dss" is as above, but indicates that the key is a DSS-key.

7. Key Exchange

Key exchange (kex) begins by each side sending name-lists of supported algorithms. Each side has a preferred algorithm in each category, and it is assumed that most implementations, at any given time, will use the same preferred algorithm. Each side MAY guess

which algorithm the other side is using, and MAY send an initial key exchange packet according to the algorithm, if appropriate for the preferred method.

The guess is considered wrong if:

- the kex algorithm and/or the host key algorithm is guessed wrong (server and client have different preferred algorithm), or
- if any of the other algorithms cannot be agreed upon (the procedure is defined below in Section 7.1).

Otherwise, the guess is considered to be right, and the optimistically sent packet MUST be handled as the first key exchange packet.

However, if the guess was wrong, and a packet was optimistically sent by one or both parties, such packets MUST be ignored (even if the error in the guess would not affect the contents of the initial packet(s)), and the appropriate side MUST send the correct initial packet.

A key exchange method uses explicit server authentication if the key exchange messages include a signature or other proof of the server's authenticity. A key exchange method uses implicit server authentication if, in order to prove its authenticity, the server also has to prove that it knows the shared secret, K, by sending a message and a corresponding MAC that the client can verify.

The key exchange method defined by this document uses explicit server authentication. However, key exchange methods with implicit server authentication MAY be used with this protocol. After a key exchange with implicit server authentication, the client MUST wait for a response to its service request message before sending any further data.

7.1. Algorithm Negotiation

Key exchange begins by each side sending the following packet:

```
byte         SSH_MSG_KEXINIT
byte[16]     cookie (random bytes)
name-list    kex_algorithms
name-list    server_host_key_algorithms
name-list    encryption_algorithms_client_to_server
name-list    encryption_algorithms_server_to_client
name-list    mac_algorithms_client_to_server
name-list    mac_algorithms_server_to_client
name-list    compression_algorithms_client_to_server
name-list    compression_algorithms_server_to_client
name-list    languages_client_to_server
name-list    languages_server_to_client
boolean      first_kex_packet_follows
uint32       0 (reserved for future extension)
```

Each of the algorithm name-lists MUST be a comma-separated list of algorithm names (see Algorithm Naming in [SSH-ARCH] and additional information in [SSH-NUMBERS]). Each supported (allowed) algorithm MUST be listed in order of preference, from most to least.

The first algorithm in each name-list MUST be the preferred (guessed) algorithm. Each name-list MUST contain at least one algorithm name.

cookie
: The 'cookie' MUST be a random value generated by the sender. Its purpose is to make it impossible for either side to fully determine the keys and the session identifier.

kex_algorithms
: Key exchange algorithms were defined above. The first algorithm MUST be the preferred (and guessed) algorithm. If both sides make the same guess, that algorithm MUST be used. Otherwise, the following algorithm MUST be used to choose a key exchange method: Iterate over client's kex algorithms, one at a time. Choose the first algorithm that satisfies the following conditions:

  + the server also supports the algorithm,

  + if the algorithm requires an encryption-capable host key, there is an encryption-capable algorithm on the server's server_host_key_algorithms that is also supported by the client, and

+ if the algorithm requires a signature-capable host key, there is a signature-capable algorithm on the server's server_host_key_algorithms that is also supported by the client.

If no algorithm satisfying all these conditions can be found, the connection fails, and both sides MUST disconnect.

server_host_key_algorithms
A name-list of the algorithms supported for the server host key. The server lists the algorithms for which it has host keys; the client lists the algorithms that it is willing to accept. There MAY be multiple host keys for a host, possibly with different algorithms.

Some host keys may not support both signatures and encryption (this can be determined from the algorithm), and thus not all host keys are valid for all key exchange methods.

Algorithm selection depends on whether the chosen key exchange algorithm requires a signature or an encryption-capable host key. It MUST be possible to determine this from the public key algorithm name. The first algorithm on the client's name-list that satisfies the requirements and is also supported by the server MUST be chosen. If there is no such algorithm, both sides MUST disconnect.

encryption_algorithms
A name-list of acceptable symmetric encryption algorithms (also known as ciphers) in order of preference. The chosen encryption algorithm to each direction MUST be the first algorithm on the client's name-list that is also on the server's name-list. If there is no such algorithm, both sides MUST disconnect.

Note that "none" must be explicitly listed if it is to be acceptable. The defined algorithm names are listed in Section 6.3.

mac_algorithms
A name-list of acceptable MAC algorithms in order of preference. The chosen MAC algorithm MUST be the first algorithm on the client's name-list that is also on the server's name-list. If there is no such algorithm, both sides MUST disconnect.

Note that "none" must be explicitly listed if it is to be acceptable. The MAC algorithm names are listed in Section 6.4.

compression_algorithms
A name-list of acceptable compression algorithms in order of preference. The chosen compression algorithm MUST be the first algorithm on the client's name-list that is also on the server's name-list. If there is no such algorithm, both sides MUST disconnect.

Note that "none" must be explicitly listed if it is to be acceptable. The compression algorithm names are listed in Section 6.2.

languages
This is a name-list of language tags in order of preference [RFC3066]. Both parties MAY ignore this name-list. If there are no language preferences, this name-list SHOULD be empty as defined in Section 5 of [SSH-ARCH]. Language tags SHOULD NOT be present unless they are known to be needed by the sending party.

first_kex_packet_follows
Indicates whether a guessed key exchange packet follows. If a guessed packet will be sent, this MUST be TRUE. If no guessed packet will be sent, this MUST be FALSE.

After receiving the SSH_MSG_KEXINIT packet from the other side, each party will know whether their guess was right. If the other party's guess was wrong, and this field was TRUE, the next packet MUST be silently ignored, and both sides MUST then act as determined by the negotiated key exchange method. If the guess was right, key exchange MUST continue using the guessed packet.

After the SSH_MSG_KEXINIT message exchange, the key exchange algorithm is run. It may involve several packet exchanges, as specified by the key exchange method.

Once a party has sent a SSH_MSG_KEXINIT message for key exchange or re-exchange, until it has sent a SSH_MSG_NEWKEYS message (Section 7.3), it MUST NOT send any messages other than:

- Transport layer generic messages (1 to 19) (but SSH_MSG_SERVICE_REQUEST and SSH_MSG_SERVICE_ACCEPT MUST NOT be sent);
- Algorithm negotiation messages (20 to 29) (but further SSH_MSG_KEXINIT messages MUST NOT be sent);
- Specific key exchange method messages (30 to 49).

The provisions of Section 11 apply to unrecognized messages.

Note, however, that during a key re-exchange, after sending a SSH_MSG_KEXINIT message, each party MUST be prepared to process an arbitrary number of messages that may be in-flight before receiving a SSH_MSG_KEXINIT message from the other party.

7.2. Output from Key Exchange

The key exchange produces two values: a shared secret K, and an exchange hash H. Encryption and authentication keys are derived from these. The exchange hash H from the first key exchange is additionally used as the session identifier, which is a unique identifier for this connection. It is used by authentication methods as a part of the data that is signed as a proof of possession of a private key. Once computed, the session identifier is not changed, even if keys are later re-exchanged.

Each key exchange method specifies a hash function that is used in the key exchange. The same hash algorithm MUST be used in key derivation. Here, we'll call it HASH.

Encryption keys MUST be computed as HASH, of a known value and K, as follows:

- Initial IV client to server: HASH(K || H || "A" || session_id) (Here K is encoded as mpint and "A" as byte and session_id as raw data. "A" means the single character A, ASCII 65).

- Initial IV server to client: HASH(K || H || "B" || session_id)

- Encryption key client to server: HASH(K || H || "C" || session_id)

- Encryption key server to client: HASH(K || H || "D" || session_id)

- Integrity key client to server: HASH(K || H || "E" || session_id)

- Integrity key server to client: HASH(K || H || "F" || session_id)

Key data MUST be taken from the beginning of the hash output. As many bytes as needed are taken from the beginning of the hash value. If the key length needed is longer than the output of the HASH, the key is extended by computing HASH of the concatenation of K and H and the entire key so far, and appending the resulting bytes (as many as HASH generates) to the key. This process is repeated until enough key material is available; the key is taken from the beginning of this value. In other words:

```
K1 = HASH(K || H || X || session_id)   (X is e.g., "A")
K2 = HASH(K || H || K1)
K3 = HASH(K || H || K1 || K2)
...
key = K1 || K2 || K3 || ...
```

This process will lose entropy if the amount of entropy in K is larger than the internal state size of HASH.

7.3. Taking Keys Into Use

Key exchange ends by each side sending an SSH_MSG_NEWKEYS message. This message is sent with the old keys and algorithms. All messages sent after this message MUST use the new keys and algorithms.

When this message is received, the new keys and algorithms MUST be used for receiving.

The purpose of this message is to ensure that a party is able to respond with an SSH_MSG_DISCONNECT message that the other party can understand if something goes wrong with the key exchange.

```
byte      SSH_MSG_NEWKEYS
```

8. Diffie-Hellman Key Exchange

The Diffie-Hellman (DH) key exchange provides a shared secret that cannot be determined by either party alone. The key exchange is combined with a signature with the host key to provide host authentication. This key exchange method provides explicit server authentication as defined in Section 7.

The following steps are used to exchange a key. In this, C is the client; S is the server; p is a large safe prime; g is a generator for a subgroup of GF(p); q is the order of the subgroup; V_S is S's identification string; V_C is C's identification string; K_S is S's public host key; I_C is C's SSH_MSG_KEXINIT message and I_S is S's SSH_MSG_KEXINIT message that have been exchanged before this part begins.

1. C generates a random number x (1 < x < q) and computes e = g^x mod p. C sends e to S.

2. S generates a random number y (0 < y < q) and computes f = g^y mod p. S receives e. It computes K = e^y mod p, H = hash(V_C || V_S || I_C || I_S || K_S || e || f || K) (these elements are encoded according to their types; see below), and signature s on H with its private host key. S sends (K_S || f || s) to C. The signing operation may involve a second hashing operation.

3. C verifies that K_S really is the host key for S (e.g., using certificates or a local database). C is also allowed to accept the key without verification; however, doing so will render the protocol insecure against active attacks (but may be desirable for practical reasons in the short term in many environments). C then computes K = f^x mod p, H = hash(V_C || V_S || I_C || I_S || K_S || e || f || K), and verifies the signature s on H.

Values of 'e' or 'f' that are not in the range [1, p-1] MUST NOT be sent or accepted by either side. If this condition is violated, the key exchange fails.

This is implemented with the following messages. The hash algorithm for computing the exchange hash is defined by the method name, and is called HASH. The public key algorithm for signing is negotiated with the SSH_MSG_KEXINIT messages.

First, the client sends the following:

```
byte      SSH_MSG_KEXDH_INIT
mpint     e
```

The server then responds with the following:

```
byte      SSH_MSG_KEXDH_REPLY
string    server public host key and certificates (K_S)
mpint     f
string    signature of H
```

The hash H is computed as the HASH hash of the concatenation of the following:

string	V_C, the client's identification string (CR and LF excluded)
string	V_S, the server's identification string (CR and LF excluded)
string	I_C, the payload of the client's SSH_MSG_KEXINIT
string	I_S, the payload of the server's SSH_MSG_KEXINIT
string	K_S, the host key
mpint	e, exchange value sent by the client
mpint	f, exchange value sent by the server
mpint	K, the shared secret

This value is called the exchange hash, and it is used to authenticate the key exchange. The exchange hash SHOULD be kept secret.

The signature algorithm MUST be applied over H, not the original data. Most signature algorithms include hashing and additional padding (e.g., "ssh-dss" specifies SHA-1 hashing). In that case, the data is first hashed with HASH to compute H, and H is then hashed with SHA-1 as part of the signing operation.

8.1. diffie-hellman-group1-sha1

The "diffie-hellman-group1-sha1" method specifies the Diffie-Hellman key exchange with SHA-1 as HASH, and Oakley Group 2 [RFC2409] (1024-bit MODP Group). This method MUST be supported for interoperability as all of the known implementations currently support it. Note that this method is named using the phrase "group1", even though it specifies the use of Oakley Group 2.

8.2. diffie-hellman-group14-sha1

The "diffie-hellman-group14-sha1" method specifies a Diffie-Hellman key exchange with SHA-1 as HASH and Oakley Group 14 [RFC3526] (2048-bit MODP Group), and it MUST also be supported.

9. Key Re-Exchange

Key re-exchange is started by sending an SSH_MSG_KEXINIT packet when not already doing a key exchange (as described in Section 7.1). When this message is received, a party MUST respond with its own SSH_MSG_KEXINIT message, except when the received SSH_MSG_KEXINIT already was a reply. Either party MAY initiate the re-exchange, but roles MUST NOT be changed (i.e., the server remains the server, and the client remains the client).

Key re-exchange is performed using whatever encryption was in effect when the exchange was started. Encryption, compression, and MAC methods are not changed before a new SSH_MSG_NEWKEYS is sent after the key exchange (as in the initial key exchange). Re-exchange is processed identically to the initial key exchange, except for the session identifier that will remain unchanged. It is permissible to change some or all of the algorithms during the re-exchange. Host keys can also change. All keys and initialization vectors are recomputed after the exchange. Compression and encryption contexts are reset.

It is RECOMMENDED that the keys be changed after each gigabyte of transmitted data or after each hour of connection time, whichever comes sooner. However, since the re-exchange is a public key operation, it requires a fair amount of processing power and should not be performed too often.

More application data may be sent after the SSH_MSG_NEWKEYS packet has been sent; key exchange does not affect the protocols that lie above the SSH transport layer.

10. Service Request

After the key exchange, the client requests a service. The service is identified by a name. The format of names and procedures for defining new names are defined in [SSH-ARCH] and [SSH-NUMBERS].

Currently, the following names have been reserved:

```
ssh-userauth
ssh-connection
```

Similar local naming policy is applied to the service names, as is applied to the algorithm names. A local service should use the PRIVATE USE syntax of "servicename@domain".

```
byte      SSH_MSG_SERVICE_REQUEST
string    service name
```

If the server rejects the service request, it SHOULD send an appropriate SSH_MSG_DISCONNECT message and MUST disconnect.

When the service starts, it may have access to the session identifier generated during the key exchange.

If the server supports the service (and permits the client to use it), it MUST respond with the following:

```
byte      SSH_MSG_SERVICE_ACCEPT
string    service name
```

Message numbers used by services should be in the area reserved for them (see [SSH-ARCH] and [SSH-NUMBERS]). The transport level will continue to process its own messages.

Note that after a key exchange with implicit server authentication, the client MUST wait for a response to its service request message before sending any further data.

11. Additional Messages

Either party may send any of the following messages at any time.

11.1. Disconnection Message

```
byte      SSH_MSG_DISCONNECT
uint32    reason code
string    description in ISO-10646 UTF-8 encoding [RFC3629]
string    language tag [RFC3066]
```

This message causes immediate termination of the connection. All implementations MUST be able to process this message; they SHOULD be able to send this message.

The sender MUST NOT send or receive any data after this message, and the recipient MUST NOT accept any data after receiving this message. The Disconnection Message 'description' string gives a more specific explanation in a human-readable form. The Disconnection Message 'reason code' gives the reason in a more machine-readable format (suitable for localization), and can have the values as displayed in the table below. Note that the decimal representation is displayed in this table for readability, but the values are actually uint32 values.

Symbolic name	reason code
SSH_DISCONNECT_HOST_NOT_ALLOWED_TO_CONNECT	1
SSH_DISCONNECT_PROTOCOL_ERROR	2
SSH_DISCONNECT_KEY_EXCHANGE_FAILED	3
SSH_DISCONNECT_RESERVED	4
SSH_DISCONNECT_MAC_ERROR	5
SSH_DISCONNECT_COMPRESSION_ERROR	6
SSH_DISCONNECT_SERVICE_NOT_AVAILABLE	7
SSH_DISCONNECT_PROTOCOL_VERSION_NOT_SUPPORTED	8
SSH_DISCONNECT_HOST_KEY_NOT_VERIFIABLE	9
SSH_DISCONNECT_CONNECTION_LOST	10
SSH_DISCONNECT_BY_APPLICATION	11
SSH_DISCONNECT_TOO_MANY_CONNECTIONS	12
SSH_DISCONNECT_AUTH_CANCELLED_BY_USER	13
SSH_DISCONNECT_NO_MORE_AUTH_METHODS_AVAILABLE	14
SSH_DISCONNECT_ILLEGAL_USER_NAME	15

If the 'description' string is displayed, the control character filtering discussed in [SSH-ARCH] should be used to avoid attacks by sending terminal control characters.

Requests for assignments of new Disconnection Message 'reason code' values (and associated 'description' text) in the range of 0x00000010 to 0xFDFFFFFF MUST be done through the IETF CONSENSUS method, as described in [RFC2434]. The Disconnection Message 'reason code' values in the range of 0xFE000000 through 0xFFFFFFFF are reserved for PRIVATE USE. As noted, the actual instructions to the IANA are in [SSH-NUMBERS].

11.2. Ignored Data Message

```
byte      SSH_MSG_IGNORE
string    data
```

All implementations MUST understand (and ignore) this message at any time (after receiving the identification string). No implementation is required to send them. This message can be used as an additional protection measure against advanced traffic analysis techniques.

11.3. Debug Message

```
byte      SSH_MSG_DEBUG
boolean   always_display
string    message in ISO-10646 UTF-8 encoding [RFC3629]
string    language tag [RFC3066]
```

All implementations MUST understand this message, but they are allowed to ignore it. This message is used to transmit information that may help debugging. If 'always_display' is TRUE, the message SHOULD be displayed. Otherwise, it SHOULD NOT be displayed unless debugging information has been explicitly requested by the user.

The 'message' doesn't need to contain a newline. It is, however, allowed to consist of multiple lines separated by CRLF (Carriage Return - Line Feed) pairs.

If the 'message' string is displayed, the terminal control character filtering discussed in [SSH-ARCH] should be used to avoid attacks by sending terminal control characters.

11.4. Reserved Messages

An implementation MUST respond to all unrecognized messages with an SSH_MSG_UNIMPLEMENTED message in the order in which the messages were received. Such messages MUST be otherwise ignored. Later protocol versions may define other meanings for these message types.

```
byte      SSH_MSG_UNIMPLEMENTED
uint32    packet sequence number of rejected message
```

12. Summary of Message Numbers

The following is a summary of messages and their associated message number.

SSH_MSG_DISCONNECT	1
SSH_MSG_IGNORE	2
SSH_MSG_UNIMPLEMENTED	3
SSH_MSG_DEBUG	4
SSH_MSG_SERVICE_REQUEST	5
SSH_MSG_SERVICE_ACCEPT	6
SSH_MSG_KEXINIT	20
SSH_MSG_NEWKEYS	21

Note that numbers 30-49 are used for kex packets. Different kex methods may reuse message numbers in this range.

13. IANA Considerations

This document is part of a set. The IANA considerations for the SSH protocol as defined in [SSH-ARCH], [SSH-USERAUTH], [SSH-CONNECT], and this document, are detailed in [SSH-NUMBERS].

14. Security Considerations

This protocol provides a secure encrypted channel over an insecure network. It performs server host authentication, key exchange, encryption, and integrity protection. It also derives a unique session ID that may be used by higher-level protocols.

Full security considerations for this protocol are provided in [SSH-ARCH].

15. References

15.1. Normative References

[SSH-ARCH] Ylonen, T. and C. Lonvick, Ed., "The Secure Shell (SSH) Protocol Architecture", RFC 4251, January 2006.

[SSH-USERAUTH] Ylonen, T. and C. Lonvick, Ed., "The Secure Shell (SSH) Authentication Protocol", RFC 4252, January 2006.

[SSH-CONNECT] Ylonen, T. and C. Lonvick, Ed., "The Secure Shell (SSH) Connection Protocol", RFC 4254, January 2006.

[SSH-NUMBERS] Lehtinen, S. and C. Lonvick, Ed., "The Secure Shell (SSH) Protocol Assigned Numbers", RFC 4250, January 2006.

[RFC1321] Rivest, R., "The MD5 Message-Digest Algorithm ", RFC 1321, April 1992.

[RFC1950] Deutsch, P. and J-L. Gailly, "ZLIB Compressed Data Format Specification version 3.3", RFC 1950, May 1996.

[RFC1951] Deutsch, P., "DEFLATE Compressed Data Format Specification version 1.3", RFC 1951, May 1996.

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, February 1997.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2144] Adams, C., "The CAST-128 Encryption Algorithm", RFC 2144, May 1997.

[RFC2409] Harkins, D. and D. Carrel, "The Internet Key Exchange (IKE)", RFC 2409, November 1998.

[RFC2434] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 2434, October 1998.

[RFC2440] Callas, J., Donnerhacke, L., Finney, H., and R. Thayer, "OpenPGP Message Format", RFC 2440, November 1998.

[RFC3066] Alvestrand, H., "Tags for the Identification of Languages", BCP 47, RFC 3066, January 2001.

[RFC3447] Jonsson, J. and B. Kaliski, "Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1", RFC 3447, February 2003.

[RFC3526] Kivinen, T. and M. Kojo, "More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE)", RFC 3526, May 2003.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, November 2003.

[FIPS-180-2] US National Institute of Standards and Technology, "Secure Hash Standard (SHS)", Federal Information Processing Standards Publication 180-2, August 2002.

[FIPS-186-2] US National Institute of Standards and Technology, "Digital Signature Standard (DSS)", Federal Information Processing Standards Publication 186-2, January 2000.

[FIPS-197] US National Institute of Standards and Technology, "Advanced Encryption Standard (AES)", Federal Information Processing Standards Publication 197, November 2001.

[FIPS-46-3] US National Institute of Standards and Technology, "Data Encryption Standard (DES)", Federal Information Processing Standards Publication 46-3, October 1999.

[SCHNEIER] Schneier, B., "Applied Cryptography Second Edition: protocols algorithms and source in code in C", John Wiley and Sons, New York, NY, 1996.

[TWOFISH] Schneier, B., "The Twofish Encryptions Algorithm: A 128-Bit Block Cipher, 1st Edition", March 1999.

15.2. Informative References

[RFC0894] Hornig, C., "Standard for the transmission of IP datagrams over Ethernet networks", STD 41, RFC 894, April 1984.

[RFC1661] Simpson, W., "The Point-to-Point Protocol (PPP)", STD 51, RFC 1661, July 1994.

[RFC2412] Orman, H., "The OAKLEY Key Determination Protocol", RFC 2412, November 1998.

[ssh-1.2.30] Ylonen, T., "ssh-1.2.30/RFC", File within compressed tarball ftp://ftp.funet.fi/pub/unix/security/login/ssh/ssh-1.2.30.tar.gz, November 1995.

Authors' Addresses

Tatu Ylonen
SSH Communications Security Corp
Valimotie 17
00380 Helsinki
Finland

EMail: ylo@ssh.com

Chris Lonvick (editor)
Cisco Systems, Inc.
12515 Research Blvd.
Austin 78759
USA

EMail: clonvick@cisco.com

Trademark Notice

Full Copyright Statement

Intellectual Property

Acknowledgement

Funding for the RFC Editor function is provided by the IETF Administrative Support Activity (IASA).

Exhibit 4

Network Working Group
Request for Comments: 4301
Obsoletes: 2401
Category: Standards Track

S. Kent
K. Seo
BBN Technologies

December 2005

Security Architecture for the Internet Protocol

Status of This Memo

Copyright Notice

Abstract

This document describes an updated version of the "Security Architecture for IP", which is designed to provide security services for traffic at the IP layer. This document obsoletes RFC 2401 (November 1998).

Dedication

This document is dedicated to the memory of Charlie Lynn, a long-time senior colleague at BBN, who made very significant contributions to the IPsec documents.

Table of Contents

1. Introduction

1.1. Summary of Contents of Document

This document specifies the base architecture for IPsec-compliant systems. It describes how to provide a set of security services for traffic at the IP layer, in both the IPv4 [Pos81a] and IPv6 [DH98] environments. This document describes the requirements for systems that implement IPsec, the fundamental elements of such systems, and how the elements fit together and fit into the IP environment. It also describes the security services offered by the IPsec protocols, and how these services can be employed in the IP environment. This document does not address all aspects of the IPsec architecture. Other documents address additional architectural details in specialized environments, e.g., use of IPsec in Network Address Translation (NAT) environments and more comprehensive support for IP multicast. The fundamental components of the IPsec security architecture are discussed in terms of their underlying, required functionality. Additional RFCs (see Section 1.3 for pointers to other documents) define the protocols in (a), (c), and (d).

a. Security Protocols -- Authentication Header (AH) and Encapsulating Security Payload (ESP)
b. Security Associations -- what they are and how they work, how they are managed, associated processing
c. Key Management -- manual and automated (The Internet Key Exchange (IKE))
d. Cryptographic algorithms for authentication and encryption

This document is not a Security Architecture for the Internet; it addresses security only at the IP layer, provided through the use of a combination of cryptographic and protocol security mechanisms.

The spelling "IPsec" is preferred and used throughout this and all related IPsec standards. All other capitalizations of IPsec (e.g., IPSEC, IPSec, ipsec) are deprecated. However, any capitalization of the sequence of letters "IPsec" should be understood to refer to the IPsec protocols.

The keywords MUST, MUST NOT, REQUIRED, SHALL, SHALL NOT, SHOULD, SHOULD NOT, RECOMMENDED, MAY, and OPTIONAL, when they appear in this document, are to be interpreted as described in RFC 2119 [Bra97].

1.2. Audience

The target audience for this document is primarily individuals who implement this IP security technology or who architect systems that will use this technology. Technically adept users of this technology

(end users or system administrators) also are part of the target audience. A glossary is provided in Appendix A to help fill in gaps in background/vocabulary. This document assumes that the reader is familiar with the Internet Protocol (IP), related networking technology, and general information system security terms and concepts.

1.3. Related Documents

As mentioned above, other documents provide detailed definitions of some of the components of IPsec and of their interrelationship. They include RFCs on the following topics:

a. security protocols -- RFCs describing the Authentication Header (AH) [Ken05b] and Encapsulating Security Payload (ESP) [Ken05a] protocols.
b. cryptographic algorithms for integrity and encryption -- one RFC that defines the mandatory, default algorithms for use with AH and ESP [Eas05], a similar RFC that defines the mandatory algorithms for use with IKEv2 [Sch05] plus a separate RFC for each cryptographic algorithm.
c. automatic key management -- RFCs on "The Internet Key Exchange (IKEv2) Protocol" [Kau05] and "Cryptographic Algorithms for Use in the Internet Key Exchange Version 2 (IKEv2)" [Sch05].

2. Design Objectives

2.1. Goals/Objectives/Requirements/Problem Description

IPsec is designed to provide interoperable, high quality, cryptographically-based security for IPv4 and IPv6. The set of security services offered includes access control, connectionless integrity, data origin authentication, detection and rejection of replays (a form of partial sequence integrity), confidentiality (via encryption), and limited traffic flow confidentiality. These services are provided at the IP layer, offering protection in a standard fashion for all protocols that may be carried over IP (including IP itself).

IPsec includes a specification for minimal firewall functionality, since that is an essential aspect of access control at the IP layer. Implementations are free to provide more sophisticated firewall mechanisms, and to implement the IPsec-mandated functionality using those more sophisticated mechanisms. (Note that interoperability may suffer if additional firewall constraints on traffic flows are imposed by an IPsec implementation but cannot be negotiated based on the traffic selector features defined in this document and negotiated

via IKEv2.) The IPsec firewall function makes use of the cryptographically-enforced authentication and integrity provided for all IPsec traffic to offer better access control than could be obtained through use of a firewall (one not privy to IPsec internal parameters) plus separate cryptographic protection.

Most of the security services are provided through use of two traffic security protocols, the Authentication Header (AH) and the Encapsulating Security Payload (ESP), and through the use of cryptographic key management procedures and protocols. The set of IPsec protocols employed in a context, and the ways in which they are employed, will be determined by the users/administrators in that context. It is the goal of the IPsec architecture to ensure that compliant implementations include the services and management interfaces needed to meet the security requirements of a broad user population.

When IPsec is correctly implemented and deployed, it ought not adversely affect users, hosts, and other Internet components that do not employ IPsec for traffic protection. IPsec security protocols (AH and ESP, and to a lesser extent, IKE) are designed to be cryptographic algorithm independent. This modularity permits selection of different sets of cryptographic algorithms as appropriate, without affecting the other parts of the implementation. For example, different user communities may select different sets of cryptographic algorithms (creating cryptographically-enforced cliques) if required.

To facilitate interoperability in the global Internet, a set of default cryptographic algorithms for use with AH and ESP is specified in [Eas05] and a set of mandatory-to-implement algorithms for IKEv2 is specified in [Sch05]. [Eas05] and [Sch05] will be periodically updated to keep pace with computational and cryptologic advances. By specifying these algorithms in documents that are separate from the AH, ESP, and IKEv2 specifications, these algorithms can be updated or replaced without affecting the standardization progress of the rest of the IPsec document suite. The use of these cryptographic algorithms, in conjunction with IPsec traffic protection and key management protocols, is intended to permit system and application developers to deploy high quality, Internet-layer, cryptographic security technology.

2.2. Caveats and Assumptions

The suite of IPsec protocols and associated default cryptographic algorithms are designed to provide high quality security for Internet traffic. However, the security offered by use of these protocols ultimately depends on the quality of their implementation, which is

outside the scope of this set of standards. Moreover, the security of a computer system or network is a function of many factors, including personnel, physical, procedural, compromising emanations, and computer security practices. Thus, IPsec is only one part of an overall system security architecture.

Finally, the security afforded by the use of IPsec is critically dependent on many aspects of the operating environment in which the IPsec implementation executes. For example, defects in OS security, poor quality of random number sources, sloppy system management protocols and practices, etc., can all degrade the security provided by IPsec. As above, none of these environmental attributes are within the scope of this or other IPsec standards.

3. System Overview

This section provides a high level description of how IPsec works, the components of the system, and how they fit together to provide the security services noted above. The goal of this description is to enable the reader to "picture" the overall process/system, see how it fits into the IP environment, and to provide context for later sections of this document, which describe each of the components in more detail.

An IPsec implementation operates in a host, as a security gateway (SG), or as an independent device, affording protection to IP traffic. (A security gateway is an intermediate system implementing IPsec, e.g., a firewall or router that has been IPsec-enabled.) More detail on these classes of implementations is provided later, in Section 3.3. The protection offered by IPsec is based on requirements defined by a Security Policy Database (SPD) established and maintained by a user or system administrator, or by an application operating within constraints established by either of the above. In general, packets are selected for one of three processing actions based on IP and next layer header information ("Selectors", Section 4.4.1.1) matched against entries in the SPD. Each packet is either PROTECTed using IPsec security services, DISCARDed, or allowed to BYPASS IPsec protection, based on the applicable SPD policies identified by the Selectors.

3.1. What IPsec Does

IPsec creates a boundary between unprotected and protected interfaces, for a host or a network (see Figure 1 below). Traffic traversing the boundary is subject to the access controls specified by the user or administrator responsible for the IPsec configuration. These controls indicate whether packets cross the boundary unimpeded, are afforded security services via AH or ESP, or are discarded.

IPsec security services are offered at the IP layer through selection of appropriate security protocols, cryptographic algorithms, and cryptographic keys. IPsec can be used to protect one or more "paths" (a) between a pair of hosts, (b) between a pair of security gateways, or (c) between a security gateway and a host. A compliant host implementation MUST support (a) and (c) and a compliant security gateway must support all three of these forms of connectivity, since under certain circumstances a security gateway acts as a host.

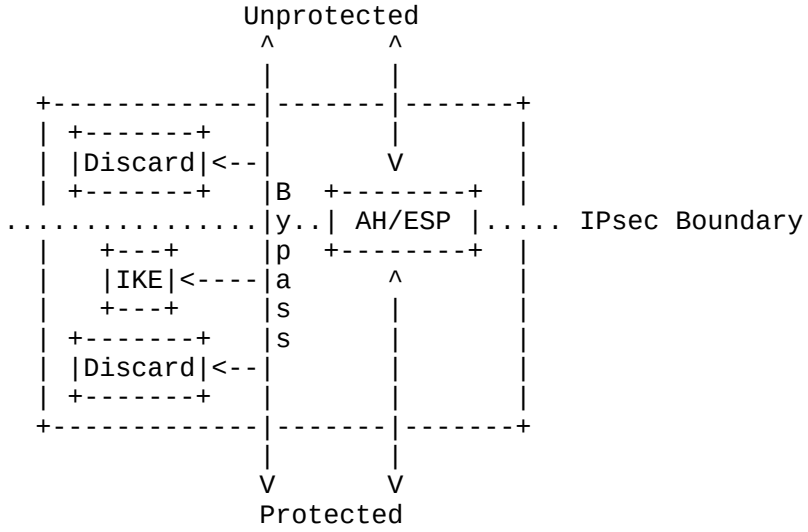


Figure 1. Top Level IPsec Processing Model

In this diagram, "unprotected" refers to an interface that might also be described as "black" or "ciphertext". Here, "protected" refers to an interface that might also be described as "red" or "plaintext". The protected interface noted above may be internal, e.g., in a host implementation of IPsec, the protected interface may link to a socket layer interface presented by the OS. In this document, the term "inbound" refers to traffic entering an IPsec implementation via the unprotected interface or emitted by the implementation on the unprotected side of the boundary and directed towards the protected interface. The term "outbound" refers to traffic entering the implementation via the protected interface, or emitted by the implementation on the protected side of the boundary and directed toward the unprotected interface. An IPsec implementation may support more than one interface on either or both sides of the boundary.

Note the facilities for discarding traffic on either side of the IPsec boundary, the BYPASS facility that allows traffic to transit the boundary without cryptographic protection, and the reference to IKE as a protected-side key and security management function.

IPsec optionally supports negotiation of IP compression [SMPT01], motivated in part by the observation that when encryption is employed within IPsec, it prevents effective compression by lower protocol layers.

3.2. How IPsec Works

IPsec uses two protocols to provide traffic security services -- Authentication Header (AH) and Encapsulating Security Payload (ESP). Both protocols are described in detail in their respective RFCs [Ken05b, Ken05a]. IPsec implementations MUST support ESP and MAY support AH. (Support for AH has been downgraded to MAY because experience has shown that there are very few contexts in which ESP cannot provide the requisite security services. Note that ESP can be used to provide only integrity, without confidentiality, making it comparable to AH in most contexts.)

- The IP Authentication Header (AH) [Ken05b] offers integrity and data origin authentication, with optional (at the discretion of the receiver) anti-replay features.

- The Encapsulating Security Payload (ESP) protocol [Ken05a] offers the same set of services, and also offers confidentiality. Use of ESP to provide confidentiality without integrity is NOT RECOMMENDED. When ESP is used with confidentiality enabled, there are provisions for limited traffic flow confidentiality, i.e., provisions for concealing packet length, and for facilitating efficient generation and discard of dummy packets. This capability is likely to be effective primarily in virtual private network (VPN) and overlay network contexts.

- Both AH and ESP offer access control, enforced through the distribution of cryptographic keys and the management of traffic flows as dictated by the Security Policy Database (SPD, Section 4.4.1).

These protocols may be applied individually or in combination with each other to provide IPv4 and IPv6 security services. However, most security requirements can be met through the use of ESP by itself. Each protocol supports two modes of use: transport mode and tunnel mode. In transport mode, AH and ESP provide protection primarily for

next layer protocols; in tunnel mode, AH and ESP are applied to tunneled IP packets. The differences between the two modes are discussed in Section 4.1.

IPsec allows the user (or system administrator) to control the granularity at which a security service is offered. For example, one can create a single encrypted tunnel to carry all the traffic between two security gateways, or a separate encrypted tunnel can be created for each TCP connection between each pair of hosts communicating across these gateways. IPsec, through the SPD management paradigm, incorporates facilities for specifying:

- which security protocol (AH or ESP) to employ, the mode (transport or tunnel), security service options, what cryptographic algorithms to use, and in what combinations to use the specified protocols and services, and

- the granularity at which protection should be applied.

Because most of the security services provided by IPsec require the use of cryptographic keys, IPsec relies on a separate set of mechanisms for putting these keys in place. This document requires support for both manual and automated distribution of keys. It specifies a specific public-key based approach (IKEv2 [Kau05]) for automated key management, but other automated key distribution techniques MAY be used.

Note: This document mandates support for several features for which support is available in IKEv2 but not in IKEv1, e.g., negotiation of an SA representing ranges of local and remote ports or negotiation of multiple SAs with the same selectors. Therefore, this document assumes use of IKEv2 or a key and security association management system with comparable features.

3.3. Where IPsec Can Be Implemented

There are many ways in which IPsec may be implemented in a host, or in conjunction with a router or firewall to create a security gateway, or as an independent security device.

a. IPsec may be integrated into the native IP stack. This requires access to the IP source code and is applicable to both hosts and security gateways, although native host implementations benefit the most from this strategy, as explained later (Section 4.4.1, paragraph 6; Section 4.4.1.1, last paragraph).

b. In a "bump-in-the-stack" (BITS) implementation, IPsec is implemented "underneath" an existing implementation of an IP protocol stack, between the native IP and the local network drivers. Source code access for the IP stack is not required in this context, making this implementation approach appropriate for use with legacy systems. This approach, when it is adopted, is usually employed in hosts.

c. The use of a dedicated, inline security protocol processor is a common design feature of systems used by the military, and of some commercial systems as well. It is sometimes referred to as a "bump-in-the-wire" (BITW) implementation. Such implementations may be designed to serve either a host or a gateway. Usually, the BITW device is itself IP addressable. When supporting a single host, it may be quite analogous to a BITS implementation, but in supporting a router or firewall, it must operate like a security gateway.

This document often talks in terms of use of IPsec by a host or a security gateway, without regard to whether the implementation is native, BITS, or BITW. When the distinctions among these implementation options are significant, the document makes reference to specific implementation approaches.

A host implementation of IPsec may appear in devices that might not be viewed as "hosts". For example, a router might employ IPsec to protect routing protocols (e.g., BGP) and management functions (e.g., Telnet), without affecting subscriber traffic traversing the router. A security gateway might employ separate IPsec implementations to protect its management traffic and subscriber traffic. The architecture described in this document is very flexible. For example, a computer with a full-featured, compliant, native OS IPsec implementation should be capable of being configured to protect resident (host) applications and to provide security gateway protection for traffic traversing the computer. Such configuration would make use of the forwarding tables and the SPD selection function described in Sections 5.1 and 5.2.

4. Security Associations

This section defines Security Association management requirements for all IPv6 implementations and for those IPv4 implementations that implement AH, ESP, or both AH and ESP. The concept of a "Security Association" (SA) is fundamental to IPsec. Both AH and ESP make use of SAs, and a major function of IKE is the establishment and maintenance of SAs. All implementations of AH or ESP MUST support the concept of an SA as described below. The remainder of this

section describes various aspects of SA management, defining required characteristics for SA policy management and SA management techniques.

4.1. Definition and Scope

An SA is a simplex "connection" that affords security services to the traffic carried by it. Security services are afforded to an SA by the use of AH, or ESP, but not both. If both AH and ESP protection are applied to a traffic stream, then two SAs must be created and coordinated to effect protection through iterated application of the security protocols. To secure typical, bi-directional communication between two IPsec-enabled systems, a pair of SAs (one in each direction) is required. IKE explicitly creates SA pairs in recognition of this common usage requirement.

For an SA used to carry unicast traffic, the Security Parameters Index (SPI) by itself suffices to specify an SA. (For information on the SPI, see Appendix A and the AH and ESP specifications [Ken05b, Ken05a].) However, as a local matter, an implementation may choose to use the SPI in conjunction with the IPsec protocol type (AH or ESP) for SA identification. If an IPsec implementation supports multicast, then it MUST support multicast SAs using the algorithm below for mapping inbound IPsec datagrams to SAs. Implementations that support only unicast traffic need not implement this de-multiplexing algorithm.

In many secure multicast architectures, e.g., [RFC3740], a central Group Controller/Key Server unilaterally assigns the Group Security Association's (GSA's) SPI. This SPI assignment is not negotiated or coordinated with the key management (e.g., IKE) subsystems that reside in the individual end systems that constitute the group. Consequently, it is possible that a GSA and a unicast SA can simultaneously use the same SPI. A multicast-capable IPsec implementation MUST correctly de-multiplex inbound traffic even in the context of SPI collisions.

Each entry in the SA Database (SAD) (Section 4.4.2) must indicate whether the SA lookup makes use of the destination IP address, or the destination and source IP addresses, in addition to the SPI. For multicast SAs, the protocol field is not employed for SA lookups. For each inbound, IPsec-protected packet, an implementation must conduct its search of the SAD such that it finds the entry that matches the "longest" SA identifier. In this context, if two or more SAD entries match based on the SPI value, then the entry that also matches based on destination address, or destination and source address (as indicated in the SAD entry) is the "longest" match. This implies a logical ordering of the SAD search as follows:

1. Search the SAD for a match on the combination of SPI, destination address, and source address. If an SAD entry matches, then process the inbound packet with that matching SAD entry. Otherwise, proceed to step 2.

2. Search the SAD for a match on both SPI and destination address. If the SAD entry matches, then process the inbound packet with that matching SAD entry. Otherwise, proceed to step 3.

3. Search the SAD for a match on only SPI if the receiver has chosen to maintain a single SPI space for AH and ESP, and on both SPI and protocol, otherwise. If an SAD entry matches, then process the inbound packet with that matching SAD entry. Otherwise, discard the packet and log an auditable event.

In practice, an implementation may choose any method (or none at all) to accelerate this search, although its externally visible behavior MUST be functionally equivalent to having searched the SAD in the above order. For example, a software-based implementation could index into a hash table by the SPI. The SAD entries in each hash table bucket's linked list could be kept sorted to have those SAD entries with the longest SA identifiers first in that linked list. Those SAD entries having the shortest SA identifiers could be sorted so that they are the last entries in the linked list. A hardware-based implementation may be able to effect the longest match search intrinsically, using commonly available Ternary Content-Addressable Memory (TCAM) features.

The indication of whether source and destination address matching is required to map inbound IPsec traffic to SAs MUST be set either as a side effect of manual SA configuration or via negotiation using an SA management protocol, e.g., IKE or Group Domain of Interpretation (GDOI) [RFC3547]. Typically, Source-Specific Multicast (SSM) [HC03] groups use a 3-tuple SA identifier composed of an SPI, a destination multicast address, and source address. An Any-Source Multicast group SA requires only an SPI and a destination multicast address as an identifier.

If different classes of traffic (distinguished by Differentiated Services Code Point (DSCP) bits [NiBlBaBL98], [Gro02]) are sent on the same SA, and if the receiver is employing the optional anti-replay feature available in both AH and ESP, this could result in inappropriate discarding of lower priority packets due to the windowing mechanism used by this feature. Therefore, a sender SHOULD put traffic of different classes, but with the same selector values, on different SAs to support Quality of Service (QoS) appropriately. To permit this, the IPsec implementation MUST permit establishment and maintenance of multiple SAs between a given sender and receiver,

with the same selectors. Distribution of traffic among these parallel SAs to support QoS is locally determined by the sender and is not negotiated by IKE. The receiver MUST process the packets from the different SAs without prejudice. These requirements apply to both transport and tunnel mode SAs. In the case of tunnel mode SAs, the DSCP values in question appear in the inner IP header. In transport mode, the DSCP value might change en route, but this should not cause problems with respect to IPsec processing since the value is not employed for SA selection and MUST NOT be checked as part of SA/packet validation. However, if significant re-ordering of packets occurs in an SA, e.g., as a result of changes to DSCP values en route, this may trigger packet discarding by a receiver due to application of the anti-replay mechanism.

DISCUSSION: Although the DSCP [NiBlBaBL98, Gro02] and Explicit Congestion Notification (ECN) [RaFlBl01] fields are not "selectors", as that term in used in this architecture, the sender will need a mechanism to direct packets with a given (set of) DSCP values to the appropriate SA. This mechanism might be termed a "classifier".

As noted above, two types of SAs are defined: transport mode and tunnel mode. IKE creates pairs of SAs, so for simplicity, we choose to require that both SAs in a pair be of the same mode, transport or tunnel.

A transport mode SA is an SA typically employed between a pair of hosts to provide end-to-end security services. When security is desired between two intermediate systems along a path (vs. end-to-end use of IPsec), transport mode MAY be used between security gateways or between a security gateway and a host. In the case where transport mode is used between security gateways or between a security gateway and a host, transport mode may be used to support in-IP tunneling (e.g., IP-in-IP [Per96] or Generic Routing Encapsulation (GRE) tunneling [FaLiHaMeTr00] or dynamic routing [ToEgWa04]) over transport mode SAs. To clarify, the use of transport mode by an intermediate system (e.g., a security gateway) is permitted only when applied to packets whose source address (for outbound packets) or destination address (for inbound packets) is an address belonging to the intermediate system itself. The access control functions that are an important part of IPsec are significantly limited in this context, as they cannot be applied to the end-to-end headers of the packets that traverse a transport mode SA used in this fashion. Thus, this way of using transport mode should be evaluated carefully before being employed in a specific context.

In IPv4, a transport mode security protocol header appears immediately after the IP header and any options, and before any next layer protocols (e.g., TCP or UDP). In IPv6, the security protocol header appears after the base IP header and selected extension headers, but may appear before or after destination options; it MUST appear before next layer protocols (e.g., TCP, UDP, Stream Control Transmission Protocol (SCTP)). In the case of ESP, a transport mode SA provides security services only for these next layer protocols, not for the IP header or any extension headers preceding the ESP header. In the case of AH, the protection is also extended to selected portions of the IP header preceding it, selected portions of extension headers, and selected options (contained in the IPv4 header, IPv6 Hop-by-Hop extension header, or IPv6 Destination extension headers). For more details on the coverage afforded by AH, see the AH specification [Ken05b].

A tunnel mode SA is essentially an SA applied to an IP tunnel, with the access controls applied to the headers of the traffic inside the tunnel. Two hosts MAY establish a tunnel mode SA between themselves. Aside from the two exceptions below, whenever either end of a security association is a security gateway, the SA MUST be tunnel mode. Thus, an SA between two security gateways is typically a tunnel mode SA, as is an SA between a host and a security gateway. The two exceptions are as follows.

- Where traffic is destined for a security gateway, e.g., Simple Network Management Protocol (SNMP) commands, the security gateway is acting as a host and transport mode is allowed. In this case, the SA terminates at a host (management) function within a security gateway and thus merits different treatment.

- As noted above, security gateways MAY support a transport mode SA to provide security for IP traffic between two intermediate systems along a path, e.g., between a host and a security gateway or between two security gateways.

Several concerns motivate the use of tunnel mode for an SA involving a security gateway. For example, if there are multiple paths (e.g., via different security gateways) to the same destination behind a security gateway, it is important that an IPsec packet be sent to the security gateway with which the SA was negotiated. Similarly, a packet that might be fragmented en route must have all the fragments delivered to the same IPsec instance for reassembly prior to cryptographic processing. Also, when a fragment is processed by IPsec and transmitted, then fragmented en route, it is critical that there be inner and outer headers to retain the fragmentation state data for the pre- and post-IPsec packet formats. Hence there are several reasons for employing tunnel mode when either end of an SA is

a security gateway. (Use of an IP-in-IP tunnel in conjunction with transport mode can also address these fragmentation issues. However, this configuration limits the ability of IPsec to enforce access control policies on traffic.)

Note: AH and ESP cannot be applied using transport mode to IPv4 packets that are fragments. Only tunnel mode can be employed in such cases. For IPv6, it would be feasible to carry a plaintext fragment on a transport mode SA; however, for simplicity, this restriction also applies to IPv6 packets. See Section 7 for more details on handling plaintext fragments on the protected side of the IPsec barrier.

For a tunnel mode SA, there is an "outer" IP header that specifies the IPsec processing source and destination, plus an "inner" IP header that specifies the (apparently) ultimate source and destination for the packet. The security protocol header appears after the outer IP header, and before the inner IP header. If AH is employed in tunnel mode, portions of the outer IP header are afforded protection (as above), as well as all of the tunneled IP packet (i.e., all of the inner IP header is protected, as well as next layer protocols). If ESP is employed, the protection is afforded only to the tunneled packet, not to the outer header.

In summary,

a) A host implementation of IPsec MUST support both transport and tunnel mode. This is true for native, BITS, and BITW implementations for hosts.

b) A security gateway MUST support tunnel mode and MAY support transport mode. If it supports transport mode, that should be used only when the security gateway is acting as a host, e.g., for network management, or to provide security between two intermediate systems along a path.

4.2. SA Functionality

The set of security services offered by an SA depends on the security protocol selected, the SA mode, the endpoints of the SA, and the election of optional services within the protocol.

For example, both AH and ESP offer integrity and authentication services, but the coverage differs for each protocol and differs for transport vs. tunnel mode. If the integrity of an IPv4 option or IPv6 extension header must be protected en route between sender and receiver, AH can provide this service, except for IP or extension headers that may change in a fashion not predictable by the sender.

However, the same security may be achieved in some contexts by applying ESP to a tunnel carrying a packet.

The granularity of access control provided is determined by the choice of the selectors that define each SA. Moreover, the authentication means employed by IPsec peers, e.g., during creation of an IKE (vs. child) SA also affects the granularity of the access control afforded.

If confidentiality is selected, then an ESP (tunnel mode) SA between two security gateways can offer partial traffic flow confidentiality. The use of tunnel mode allows the inner IP headers to be encrypted, concealing the identities of the (ultimate) traffic source and destination. Moreover, ESP payload padding also can be invoked to hide the size of the packets, further concealing the external characteristics of the traffic. Similar traffic flow confidentiality services may be offered when a mobile user is assigned a dynamic IP address in a dialup context, and establishes a (tunnel mode) ESP SA to a corporate firewall (acting as a security gateway). Note that fine-granularity SAs generally are more vulnerable to traffic analysis than coarse-granularity ones that are carrying traffic from many subscribers.

Note: A compliant implementation MUST NOT allow instantiation of an ESP SA that employs both NULL encryption and no integrity algorithm. An attempt to negotiate such an SA is an auditable event by both initiator and responder. The audit log entry for this event SHOULD include the current date/time, local IKE IP address, and remote IKE IP address. The initiator SHOULD record the relevant SPD entry.

4.3. Combining SAs

This document does not require support for nested security associations or for what RFC 2401 [RFC2401] called "SA bundles". These features still can be effected by appropriate configuration of both the SPD and the local forwarding functions (for inbound and outbound traffic), but this capability is outside of the IPsec module and thus the scope of this specification. As a result, management of nested/bundled SAs is potentially more complex and less assured than under the model implied by RFC 2401 [RFC2401]. An implementation that provides support for nested SAs SHOULD provide a management interface that enables a user or administrator to express the nesting requirement, and then create the appropriate SPD entries and forwarding table entries to effect the requisite processing. (See Appendix E for an example of how to configure nested SAs.)

4.4. Major IPsec Databases

Many of the details associated with processing IP traffic in an IPsec implementation are largely a local matter, not subject to standardization. However, some external aspects of the processing must be standardized to ensure interoperability and to provide a minimum management capability that is essential for productive use of IPsec. This section describes a general model for processing IP traffic relative to IPsec functionality, in support of these interoperability and functionality goals. The model described below is nominal; implementations need not match details of this model as presented, but the external behavior of implementations MUST correspond to the externally observable characteristics of this model in order to be compliant.

There are three nominal databases in this model: the Security Policy Database (SPD), the Security Association Database (SAD), and the Peer Authorization Database (PAD). The first specifies the policies that determine the disposition of all IP traffic inbound or outbound from a host or security gateway (Section 4.4.1). The second database contains parameters that are associated with each established (keyed) SA (Section 4.4.2). The third database, the PAD, provides a link between an SA management protocol (such as IKE) and the SPD (Section 4.4.3).

Multiple Separate IPsec Contexts

If an IPsec implementation acts as a security gateway for multiple subscribers, it MAY implement multiple separate IPsec contexts. Each context MAY have and MAY use completely independent identities, policies, key management SAs, and/or IPsec SAs. This is for the most part a local implementation matter. However, a means for associating inbound (SA) proposals with local contexts is required. To this end, if supported by the key management protocol in use, context identifiers MAY be conveyed from initiator to responder in the signaling messages, with the result that IPsec SAs are created with a binding to a particular context. For example, a security gateway that provides VPN service to multiple customers will be able to associate each customer’s traffic with the correct VPN.

Forwarding vs Security Decisions

The IPsec model described here embodies a clear separation between forwarding (routing) and security decisions, to accommodate a wide range of contexts where IPsec may be employed. Forwarding may be trivial, in the case where there are only two interfaces, or it may be complex, e.g., if the context in which IPsec is implemented

employs a sophisticated forwarding function. IPsec assumes only that outbound and inbound traffic that has passed through IPsec processing is forwarded in a fashion consistent with the context in which IPsec is implemented. Support for nested SAs is optional; if required, it requires coordination between forwarding tables and SPD entries to cause a packet to traverse the IPsec boundary more than once.

"Local" vs "Remote"

In this document, with respect to IP addresses and ports, the terms "Local" and "Remote" are used for policy rules. "Local" refers to the entity being protected by an IPsec implementation, i.e., the "source" address/port of outbound packets or the "destination" address/port of inbound packets. "Remote" refers to a peer entity or peer entities. The terms "source" and "destination" are used for packet header fields.

"Non-initial" vs "Initial" Fragments

Throughout this document, the phrase "non-initial fragments" is used to mean fragments that do not contain all of the selector values that may be needed for access control (e.g., they might not contain Next Layer Protocol, source and destination ports, ICMP message type/code, Mobility Header type). And the phrase "initial fragment" is used to mean a fragment that contains all the selector values needed for access control. However, it should be noted that for IPv6, which fragment contains the Next Layer Protocol and ports (or ICMP message type/code or Mobility Header type [Mobip]) will depend on the kind and number of extension headers present. The "initial fragment" might not be the first fragment, in this context.

4.4.1. The Security Policy Database (SPD)

An SA is a management construct used to enforce security policy for traffic crossing the IPsec boundary. Thus, an essential element of SA processing is an underlying Security Policy Database (SPD) that specifies what services are to be offered to IP datagrams and in what fashion. The form of the database and its interface are outside the scope of this specification. However, this section specifies minimum management functionality that must be provided, to allow a user or system administrator to control whether and how IPsec is applied to traffic transmitted or received by a host or transiting a security gateway. The SPD, or relevant caches, must be consulted during the processing of all traffic (inbound and outbound), including traffic not protected by IPsec, that traverses the IPsec boundary. This includes IPsec management traffic such as IKE. An IPsec

implementation MUST have at least one SPD, and it MAY support multiple SPDs, if appropriate for the context in which the IPsec implementation operates. There is no requirement to maintain SPDs on a per-interface basis, as was specified in RFC 2401 [RFC2401]. However, if an implementation supports multiple SPDs, then it MUST include an explicit SPD selection function that is invoked to select the appropriate SPD for outbound traffic processing. The inputs to this function are the outbound packet and any local metadata (e.g., the interface via which the packet arrived) required to effect the SPD selection function. The output of the function is an SPD identifier (SPD-ID).

The SPD is an ordered database, consistent with the use of Access Control Lists (ACLs) or packet filters in firewalls, routers, etc. The ordering requirement arises because entries often will overlap due to the presence of (non-trivial) ranges as values for selectors. Thus, a user or administrator MUST be able to order the entries to express a desired access control policy. There is no way to impose a general, canonical order on SPD entries, because of the allowed use of wildcards for selector values and because the different types of selectors are not hierarchically related.

Processing Choices: DISCARD, BYPASS, PROTECT

An SPD must discriminate among traffic that is afforded IPsec protection and traffic that is allowed to bypass IPsec. This applies to the IPsec protection to be applied by a sender and to the IPsec protection that must be present at the receiver. For any outbound or inbound datagram, three processing choices are possible: DISCARD, BYPASS IPsec, or PROTECT using IPsec. The first choice refers to traffic that is not allowed to traverse the IPsec boundary (in the specified direction). The second choice refers to traffic that is allowed to cross the IPsec boundary without IPsec protection. The third choice refers to traffic that is afforded IPsec protection, and for such traffic the SPD must specify the security protocols to be employed, their mode, security service options, and the cryptographic algorithms to be used.

SPD-S, SPD-I, SPD-O

An SPD is logically divided into three pieces. The SPD-S (secure traffic) contains entries for all traffic subject to IPsec protection. SPD-O (outbound) contains entries for all outbound traffic that is to be bypassed or discarded. SPD-I (inbound) is applied to inbound traffic that will be bypassed or discarded. All three of these can be decorrelated (with the exception noted above for native host implementations) to facilitate caching. If

an IPsec implementation supports only one SPD, then the SPD consists of all three parts. If multiple SPDs are supported, some of them may be partial, e.g., some SPDs might contain only SPD-I entries, to control inbound bypassed traffic on a per-interface basis. The split allows SPD-I to be consulted without having to consult SPD-S, for such traffic. Since the SPD-I is just a part of the SPD, if a packet that is looked up in the SPD-I cannot be matched to an entry there, then the packet MUST be discarded. Note that for outbound traffic, if a match is not found in SPD-S, then SPD-O must be checked to see if the traffic should be bypassed. Similarly, if SPD-O is checked first and no match is found, then SPD-S must be checked. In an ordered, non-decorrelated SPD, the entries for the SPD-S, SPD-I, and SPD-O are interleaved. So there is one lookup in the SPD.

SPD Entries

Each SPD entry specifies packet disposition as BYPASS, DISCARD, or PROTECT. The entry is keyed by a list of one or more selectors. The SPD contains an ordered list of these entries. The required selector types are defined in Section 4.4.1.1. These selectors are used to define the granularity of the SAs that are created in response to an outbound packet or in response to a proposal from a peer. The detailed structure of an SPD entry is described in Section 4.4.1.2. Every SPD SHOULD have a nominal, final entry that matches anything that is otherwise unmatched, and discards it.

The SPD MUST permit a user or administrator to specify policy entries as follows:

- SPD-I: For inbound traffic that is to be bypassed or discarded, the entry consists of the values of the selectors that apply to the traffic to be bypassed or discarded.

- SPD-O: For outbound traffic that is to be bypassed or discarded, the entry consists of the values of the selectors that apply to the traffic to be bypassed or discarded.

- SPD-S: For traffic that is to be protected using IPsec, the entry consists of the values of the selectors that apply to the traffic to be protected via AH or ESP, controls on how to create SAs based on these selectors, and the parameters needed to effect this protection (e.g., algorithms, modes, etc.). Note that an SPD-S entry also contains information such as "populate from packet" (PFP) flag (see paragraphs below on "How To Derive the Values for an SAD entry") and bits indicating whether the

SA lookup makes use of the local and remote IP addresses in addition to the SPI (see AH [Ken05b] or ESP [Ken05a] specifications).

Representing Directionality in an SPD Entry

For traffic protected by IPsec, the Local and Remote address and ports in an SPD entry are swapped to represent directionality, consistent with IKE conventions. In general, the protocols that IPsec deals with have the property of requiring symmetric SAs with flipped Local/Remote IP addresses. However, for ICMP, there is often no such bi-directional authorization requirement. Nonetheless, for the sake of uniformity and simplicity, SPD entries for ICMP are specified in the same way as for other protocols. Note also that for ICMP, Mobility Header, and non-initial fragments, there are no port fields in these packets. ICMP has message type and code and Mobility Header has mobility header type. Thus, SPD entries have provisions for expressing access controls appropriate for these protocols, in lieu of the normal port field controls. For bypassed or discarded traffic, separate inbound and outbound entries are supported, e.g., to permit unidirectional flows if required.

OPAQUE and ANY

For each selector in an SPD entry, in addition to the literal values that define a match, there are two special values: ANY and OPAQUE. ANY is a wildcard that matches any value in the corresponding field of the packet, or that matches packets where that field is not present or is obscured. OPAQUE indicates that the corresponding selector field is not available for examination because it may not be present in a fragment, it does not exist for the given Next Layer Protocol, or prior application of IPsec may have encrypted the value. The ANY value encompasses the OPAQUE value. Thus, OPAQUE need be used only when it is necessary to distinguish between the case of any allowed value for a field, vs. the absence or unavailability (e.g., due to encryption) of the field.

How to Derive the Values for an SAD Entry

For each selector in an SPD entry, the entry specifies how to derive the corresponding values for a new SA Database (SAD, see Section 4.4.2) entry from those in the SPD and the packet. The goal is to allow an SAD entry and an SPD cache entry to be created based on specific selector values from the packet, or from the matching SPD entry. For outbound traffic, there are SPD-S cache entries and SPD-O cache entries. For inbound traffic not

protected by IPsec, there are SPD-I cache entries and there is the SAD, which represents the cache for inbound IPsec-protected traffic (see Section 4.4.2). If IPsec processing is specified for an entry, a "populate from packet" (PFP) flag may be asserted for one or more of the selectors in the SPD entry (Local IP address; Remote IP address; Next Layer Protocol; and, depending on Next Layer Protocol, Local port and Remote port, or ICMP type/code, or Mobility Header type). If asserted for a given selector X, the flag indicates that the SA to be created should take its value for X from the value in the packet. Otherwise, the SA should take its value(s) for X from the value(s) in the SPD entry. Note: In the non-PFP case, the selector values negotiated by the SA management protocol (e.g., IKEv2) may be a subset of those in the SPD entry, depending on the SPD policy of the peer. Also, whether a single flag is used for, e.g., source port, ICMP type/code, and Mobility Header (MH) type, or a separate flag is used for each, is a local matter.

The following example illustrates the use of the PFP flag in the context of a security gateway or a BITS/BITW implementation. Consider an SPD entry where the allowed value for Remote address is a range of IPv4 addresses: 192.0.2.1 to 192.0.2.10. Suppose an outbound packet arrives with a destination address of 192.0.2.3, and there is no extant SA to carry this packet. The value used for the SA created to transmit this packet could be either of the two values shown below, depending on what the SPD entry for this selector says is the source of the selector value:

PFP flag value for the Remote addr. selector	example of new SAD dest. address selector value
a. PFP TRUE	192.0.2.3 (one host)
b. PFP FALSE	192.0.2.1 to 192.0.2.10 (range of hosts)

Note that if the SPD entry above had a value of ANY for the Remote address, then the SAD selector value would have to be ANY for case (b), but would still be as illustrated for case (a). Thus, the PFP flag can be used to prohibit sharing of an SA, even among packets that match the same SPD entry.

Management Interface

For every IPsec implementation, there MUST be a management interface that allows a user or system administrator to manage the SPD. The interface must allow the user (or administrator) to specify the security processing to be applied to every packet that traverses the IPsec boundary. (In a native host IPsec

implementation making use of a socket interface, the SPD may not need to be consulted on a per-packet basis, as noted at the end of Section 4.4.1.1 and in Section 5.) The management interface for the SPD MUST allow creation of entries consistent with the selectors defined in Section 4.4.1.1, and MUST support (total) ordering of these entries, as seen via this interface. The SPD entries' selectors are analogous to the ACL or packet filters commonly found in a stateless firewall or packet filtering router and which are currently managed this way.

In host systems, applications MAY be allowed to create SPD entries. (The means of signaling such requests to the IPsec implementation are outside the scope of this standard.) However, the system administrator MUST be able to specify whether or not a user or application can override (default) system policies. The form of the management interface is not specified by this document and may differ for hosts vs. security gateways, and within hosts the interface may differ for socket-based vs. BITS implementations. However, this document does specify a standard set of SPD elements that all IPsec implementations MUST support.

Decorrelation

The processing model described in this document assumes the ability to decorrelate overlapping SPD entries to permit caching, which enables more efficient processing of outbound traffic in security gateways and BITS/BITW implementations. Decorrelation [CoSa04] is only a means of improving performance and simplifying the processing description. This RFC does not require a compliant implementation to make use of decorrelation. For example, native host implementations typically make use of caching implicitly because they bind SAs to socket interfaces, and thus there is no requirement to be able to decorrelate SPD entries in these implementations.

Note: Unless otherwise qualified, the use of "SPD" refers to the body of policy information in both ordered or decorrelated (unordered) state. Appendix B provides an algorithm that can be used to decorrelate SPD entries, but any algorithm that produces equivalent output may be used. Note that when an SPD entry is decorrelated all the resulting entries MUST be linked together, so that all members of the group derived from an individual, SPD entry (prior to decorrelation) can all be placed into caches and into the SAD at the same time. For example, suppose one starts with an entry A (from an ordered SPD) that when decorrelated, yields entries A1, A2, and A3. When a packet comes along that matches, say A2, and triggers the creation of an SA, the SA management protocol (e.g., IKEv2) negotiates A. And all 3

decorrelated entries, A1, A2, and A3, are placed in the appropriate SPD-S cache and linked to the SA. The intent is that use of a decorrelated SPD ought not to create more SAs than would have resulted from use of a not-decorrelated SPD.

If a decorrelated SPD is employed, there are three options for what an initiator sends to a peer via an SA management protocol (e.g., IKE). By sending the complete set of linked, decorrelated entries that were selected from the SPD, a peer is given the best possible information to enable selection of the appropriate SPD entry at its end, especially if the peer has also decorrelated its SPD. However, if a large number of decorrelated entries are linked, this may create large packets for SA negotiation, and hence fragmentation problems for the SA management protocol.

Alternatively, the original entry from the (correlated) SPD may be retained and passed to the SA management protocol. Passing the correlated SPD entry keeps the use of a decorrelated SPD a local matter, not visible to peers, and avoids possible fragmentation concerns, although it provides less precise information to a responder for matching against the responder's SPD.

An intermediate approach is to send a subset of the complete set of linked, decorrelated SPD entries. This approach can avoid the fragmentation problems cited above yet provide better information than the original, correlated entry. The major shortcoming of this approach is that it may cause additional SAs to be created later, since only a subset of the linked, decorrelated entries are sent to a peer. Implementers are free to employ any of the approaches cited above.

A responder uses the traffic selector proposals it receives via an SA management protocol to select an appropriate entry in its SPD. The intent of the matching is to select an SPD entry and create an SA that most closely matches the intent of the initiator, so that traffic traversing the resulting SA will be accepted at both ends. If the responder employs a decorrelated SPD, it SHOULD use the decorrelated SPD entries for matching, as this will generally result in creation of SAs that are more likely to match the intent of both peers. If the responder has a correlated SPD, then it SHOULD match the proposals against the correlated entries. For IKEv2, use of a decorrelated SPD offers the best opportunity for a responder to generate a "narrowed" response.

In all cases, when a decorrelated SPD is available, the decorrelated entries are used to populate the SPD-S cache. If the SPD is not decorrelated, caching is not allowed and an ordered

search of SPD MUST be performed to verify that inbound traffic arriving on an SA is consistent with the access control policy expressed in the SPD.

Handling Changes to the SPD While the System Is Running

If a change is made to the SPD while the system is running, a check SHOULD be made of the effect of this change on extant SAs. An implementation SHOULD check the impact of an SPD change on extant SAs and SHOULD provide a user/administrator with a mechanism for configuring what actions to take, e.g., delete an affected SA, allow an affected SA to continue unchanged, etc.

4.4.1.1. Selectors

An SA may be fine-grained or coarse-grained, depending on the selectors used to define the set of traffic for the SA. For example, all traffic between two hosts may be carried via a single SA, and afforded a uniform set of security services. Alternatively, traffic between a pair of hosts might be spread over multiple SAs, depending on the applications being used (as defined by the Next Layer Protocol and related fields, e.g., ports), with different security services offered by different SAs. Similarly, all traffic between a pair of security gateways could be carried on a single SA, or one SA could be assigned for each communicating host pair. The following selector parameters MUST be supported by all IPsec implementations to facilitate control of SA granularity. Note that both Local and Remote addresses should either be IPv4 or IPv6, but not a mix of address types. Also, note that the Local/Remote port selectors (and ICMP message type and code, and Mobility Header type) may be labeled as OPAQUE to accommodate situations where these fields are inaccessible due to packet fragmentation.

- Remote IP Address(es) (IPv4 or IPv6): This is a list of ranges of IP addresses (unicast, broadcast (IPv4 only)). This structure allows expression of a single IP address (via a trivial range), or a list of addresses (each a trivial range), or a range of addresses (low and high values, inclusive), as well as the most generic form of a list of ranges. Address ranges are used to support more than one remote system sharing the same SA, e.g., behind a security gateway.

- Local IP Address(es) (IPv4 or IPv6): This is a list of ranges of IP addresses (unicast, broadcast (IPv4 only)). This structure allows expression of a single IP address (via a trivial range), or a list of addresses (each a trivial range), or a range of addresses (low and high values, inclusive), as well as the most generic form of a list of ranges. Address ranges are used to

support more than one source system sharing the same SA, e.g., behind a security gateway. Local refers to the address(es) being protected by this implementation (or policy entry).

Note: The SPD does not include support for multicast address entries. To support multicast SAs, an implementation should make use of a Group SPD (GSPD) as defined in [RFC3740]. GSPD entries require a different structure, i.e., one cannot use the symmetric relationship associated with local and remote address values for unicast SAs in a multicast context. Specifically, outbound traffic directed to a multicast address on an SA would not be received on a companion, inbound SA with the multicast address as the source.

- Next Layer Protocol: Obtained from the IPv4 "Protocol" or the IPv6 "Next Header" fields. This is an individual protocol number, ANY, or for IPv6 only, OPAQUE. The Next Layer Protocol is whatever comes after any IP extension headers that are present. To simplify locating the Next Layer Protocol, there SHOULD be a mechanism for configuring which IPv6 extension headers to skip. The default configuration for which protocols to skip SHOULD include the following protocols: 0 (Hop-by-hop options), 43 (Routing Header), 44 (Fragmentation Header), and 60 (Destination Options). Note: The default list does NOT include 51 (AH) or 50 (ESP). From a selector lookup point of view, IPsec treats AH and ESP as Next Layer Protocols.

  Several additional selectors depend on the Next Layer Protocol value:

  * If the Next Layer Protocol uses two ports (as do TCP, UDP, SCTP, and others), then there are selectors for Local and Remote Ports. Each of these selectors has a list of ranges of values. Note that the Local and Remote ports may not be available in the case of receipt of a fragmented packet or if the port fields have been protected by IPsec (encrypted); thus, a value of OPAQUE also MUST be supported. Note: In a non-initial fragment, port values will not be available. If a port selector specifies a value other than ANY or OPAQUE, it cannot match packets that are non-initial fragments. If the SA requires a port value other than ANY or OPAQUE, an arriving fragment without ports MUST be discarded. (See Section 7, "Handling Fragments".)

  * If the Next Layer Protocol is a Mobility Header, then there is a selector for IPv6 Mobility Header message type (MH type) [Mobip]. This is an 8-bit value that identifies a particular mobility message. Note that the MH type may not be available

in the case of receipt of a fragmented packet. (See Section 7, "Handling Fragments".) For IKE, the IPv6 Mobility Header message type (MH type) is placed in the most significant eight bits of the 16-bit local "port" selector.

* If the Next Layer Protocol value is ICMP, then there is a 16-bit selector for the ICMP message type and code. The message type is a single 8-bit value, which defines the type of an ICMP message, or ANY. The ICMP code is a single 8-bit value that defines a specific subtype for an ICMP message. For IKE, the message type is placed in the most significant 8 bits of the 16-bit selector and the code is placed in the least significant 8 bits. This 16-bit selector can contain a single type and a range of codes, a single type and ANY code, and ANY type and ANY code. Given a policy entry with a range of Types (T-start to T-end) and a range of Codes (C-start to C-end), and an ICMP packet with Type t and Code c, an implementation MUST test for a match using

  (T-start*256) + C-start <= (t*256) + c <= (T-end*256) + C-end

  Note that the ICMP message type and code may not be available in the case of receipt of a fragmented packet. (See Section 7, "Handling Fragments".)

- Name: This is not a selector like the others above. It is not acquired from a packet. A name may be used as a symbolic identifier for an IPsec Local or Remote address. Named SPD entries are used in two ways:

  1. A named SPD entry is used by a responder (not an initiator) in support of access control when an IP address would not be appropriate for the Remote IP address selector, e.g., for "road warriors". The name used to match this field is communicated during the IKE negotiation in the ID payload. In this context, the initiator’s Source IP address (inner IP header in tunnel mode) is bound to the Remote IP address in the SAD entry created by the IKE negotiation. This address overrides the Remote IP address value in the SPD, when the SPD entry is selected in this fashion. All IPsec implementations MUST support this use of names.

  2. A named SPD entry may be used by an initiator to identify a user for whom an IPsec SA will be created (or for whom traffic may be bypassed). The initiator’s IP source address (from inner IP header in tunnel mode) is used to replace the following if and when they are created:

- local address in the SPD cache entry
- local address in the outbound SAD entry
- remote address in the inbound SAD entry

Support for this use is optional for multi-user, native host implementations and not applicable to other implementations. Note that this name is used only locally; it is not communicated by the key management protocol. Also, name forms other than those used for case 1 above (responder) are applicable in the initiator context (see below).

An SPD entry can contain both a name (or a list of names) and also values for the Local or Remote IP address.

For case 1, responder, the identifiers employed in named SPD entries are one of the following four types:

a. a fully qualified user name string (email), e.g., mozart@foo.example.com
(this corresponds to ID_RFC822_ADDR in IKEv2)

b. a fully qualified DNS name, e.g., foo.example.com
(this corresponds to ID_FQDN in IKEv2)

c. X.500 distinguished name, e.g., [WaKiHo97], CN = Stephen T. Kent, O = BBN Technologies, SP = MA, C = US
(this corresponds to ID_DER_ASN1_DN in IKEv2, after decoding)

d. a byte string
(this corresponds to Key_ID in IKEv2)

For case 2, initiator, the identifiers employed in named SPD entries are of type byte string. They are likely to be Unix UIDs, Windows security IDs, or something similar, but could also be a user name or account name. In all cases, this identifier is only of local concern and is not transmitted.

The IPsec implementation context determines how selectors are used. For example, a native host implementation typically makes use of a socket interface. When a new connection is established, the SPD can be consulted and an SA bound to the socket. Thus, traffic sent via that socket need not result in additional lookups to the SPD (SPD-O and SPD-S) cache. In contrast, a BITS, BITW, or security gateway implementation needs to look at each packet and perform an SPD-O/SPD-S cache lookup based on the selectors.

4.4.1.2. Structure of an SPD Entry

This section contains a prose description of an SPD entry. Also, Appendix C provides an example of an ASN.1 definition of an SPD entry.

This text describes the SPD in a fashion that is intended to map directly into IKE payloads to ensure that the policy required by SPD entries can be negotiated through IKE. Unfortunately, the semantics of the version of IKEv2 published concurrently with this document [Kau05] do not align precisely with those defined for the SPD. Specifically, IKEv2 does not enable negotiation of a single SA that binds multiple pairs of local and remote addresses and ports to a single SA. Instead, when multiple local and remote addresses and ports are negotiated for an SA, IKEv2 treats these not as pairs, but as (unordered) sets of local and remote values that can be arbitrarily paired. Until IKE provides a facility that conveys the semantics that are expressed in the SPD via selector sets (as described below), users MUST NOT include multiple selector sets in a single SPD entry unless the access control intent aligns with the IKE "mix and match" semantics. An implementation MAY warn users, to alert them to this problem if users create SPD entries with multiple selector sets, the syntax of which indicates possible conflicts with current IKE semantics.

The management GUI can offer the user other forms of data entry and display, e.g., the option of using address prefixes as well as ranges, and symbolic names for protocols, ports, etc. (Do not confuse the use of symbolic names in a management interface with the SPD selector "Name".) Note that Remote/Local apply only to IP addresses and ports, not to ICMP message type/code or Mobility Header type. Also, if the reserved, symbolic selector value OPAQUE or ANY is employed for a given selector type, only that value may appear in the list for that selector, and it must appear only once in the list for that selector. Note that ANY and OPAQUE are local syntax conventions -- IKEv2 negotiates these values via the ranges indicated below:

```
ANY:     start = 0        end = <max>
OPAQUE:  start = <max>    end = 0
```

An SPD is an ordered list of entries each of which contains the following fields.

- Name -- a list of IDs. This quasi-selector is optional. The forms that MUST be supported are described above in Section 4.4.1.1 under "Name".

- o PFP flags -- one per traffic selector. A given flag, e.g., for Next Layer Protocol, applies to the relevant selector across all "selector sets" (see below) contained in an SPD entry. When creating an SA, each flag specifies for the corresponding traffic selector whether to instantiate the selector from the corresponding field in the packet that triggered the creation of the SA or from the value(s) in the corresponding SPD entry (see Section 4.4.1, "How to Derive the Values for an SAD Entry"). Whether a single flag is used for, e.g., source port, ICMP type/code, and MH type, or a separate flag is used for each, is a local matter. There are PFP flags for:
  - Local Address
  - Remote Address
  - Next Layer Protocol
  - Local Port, or ICMP message type/code or Mobility Header type (depending on the next layer protocol)
  - Remote Port, or ICMP message type/code or Mobility Header type (depending on the next layer protocol)

- o One to N selector sets that correspond to the "condition" for applying a particular IPsec action. Each selector set contains:
  - Local Address
  - Remote Address
  - Next Layer Protocol
  - Local Port, or ICMP message type/code or Mobility Header type (depending on the next layer protocol)
  - Remote Port, or ICMP message type/code or Mobility Header type (depending on the next layer protocol)

  Note: The "next protocol" selector is an individual value (unlike the local and remote IP addresses) in a selector set entry. This is consistent with how IKEv2 negotiates the Traffic Selector (TS) values for an SA. It also makes sense because one may need to associate different port fields with different protocols. It is possible to associate multiple protocols (and ports) with a single SA by specifying multiple selector sets for that SA.

- o Processing info -- which action is required -- PROTECT, BYPASS, or DISCARD. There is just one action that goes with all the selector sets, not a separate action for each set. If the required processing is PROTECT, the entry contains the following information.
  - IPsec mode -- tunnel or transport

- (if tunnel mode) local tunnel address -- For a non-mobile host, if there is just one interface, this is straightforward; if there are multiple interfaces, this must be statically configured. For a mobile host, the specification of the local address is handled externally to IPsec.
- (if tunnel mode) remote tunnel address -- There is no standard way to determine this. See 4.5.3, "Locating a Security Gateway".
- Extended Sequence Number -- Is this SA using extended sequence numbers?
- stateful fragment checking -- Is this SA using stateful fragment checking? (See Section 7 for more details.)
- Bypass DF bit (T/F) -- applicable to tunnel mode SAs
- Bypass DSCP (T/F) or map to unprotected DSCP values (array) if needed to restrict bypass of DSCP values -- applicable to tunnel mode SAs
- IPsec protocol -- AH or ESP
- algorithms -- which ones to use for AH, which ones to use for ESP, which ones to use for combined mode, ordered by decreasing priority

It is a local matter as to what information is kept with regard to handling extant SAs when the SPD is changed.

4.4.1.3. More Regarding Fields Associated with Next Layer Protocols

Additional selectors are often associated with fields in the Next Layer Protocol header. A particular Next Layer Protocol can have zero, one, or two selectors. There may be situations where there aren't both local and remote selectors for the fields that are dependent on the Next Layer Protocol. The IPv6 Mobility Header has only a Mobility Header message type. AH and ESP have no further selector fields. A system may be willing to send an ICMP message type and code that it does not want to receive. In the descriptions below, "port" is used to mean a field that is dependent on the Next Layer Protocol.

A. If a Next Layer Protocol has no "port" selectors, then the Local and Remote "port" selectors are set to OPAQUE in the relevant SPD entry, e.g.,

```
Local's
  next layer protocol = AH
  "port" selector     = OPAQUE
```

Remote's
next layer protocol = AH
"port" selector = OPAQUE

B. Even if a Next Layer Protocol has only one selector, e.g., Mobility Header type, then the Local and Remote "port" selectors are used to indicate whether a system is willing to send and/or receive traffic with the specified "port" values. For example, if Mobility Headers of a specified type are allowed to be sent and received via an SA, then the relevant SPD entry would be set as follows:

Local's
next layer protocol = Mobility Header
"port" selector = Mobility Header message type

Remote's
next layer protocol = Mobility Header
"port" selector = Mobility Header message type

If Mobility Headers of a specified type are allowed to be sent but NOT received via an SA, then the relevant SPD entry would be set as follows:

Local's
next layer protocol = Mobility Header
"port" selector = Mobility Header message type

Remote's
next layer protocol = Mobility Header
"port" selector = OPAQUE

If Mobility Headers of a specified type are allowed to be received but NOT sent via an SA, then the relevant SPD entry would be set as follows:

Local's
next layer protocol = Mobility Header
"port" selector = OPAQUE

Remote's
next layer protocol = Mobility Header
"port" selector = Mobility Header message type

C. If a system is willing to send traffic with a particular "port" value but NOT receive traffic with that kind of port value, the system's traffic selectors are set as follows in the relevant SPD entry:

```
Local's
  next layer protocol = ICMP
  "port" selector     = <specific ICMP type & code>

Remote's
  next layer protocol = ICMP
  "port" selector     = OPAQUE
```

D. To indicate that a system is willing to receive traffic with a particular "port" value but NOT send that kind of traffic, the system's traffic selectors are set as follows in the relevant SPD entry:

```
Local's
  next layer protocol = ICMP
  "port" selector     = OPAQUE

Remote's
  next layer protocol = ICMP
  "port" selector     = <specific ICMP type & code>
```

For example, if a security gateway is willing to allow systems behind it to send ICMP traceroutes, but is not willing to let outside systems run ICMP traceroutes to systems behind it, then the security gateway's traffic selectors are set as follows in the relevant SPD entry:

```
Local's
  next layer protocol = 1 (ICMPv4)
  "port" selector     = 30 (traceroute)

Remote's
  next layer protocol = 1 (ICMPv4)
  "port" selector     = OPAQUE
```

4.4.2. Security Association Database (SAD)

In each IPsec implementation, there is a nominal Security Association Database (SAD), in which each entry defines the parameters associated with one SA. Each SA has an entry in the SAD. For outbound processing, each SAD entry is pointed to by entries in the SPD-S part of the SPD cache. For inbound processing, for unicast SAs, the SPI is used either alone to look up an SA or in conjunction with the IPsec protocol type. If an IPsec implementation supports multicast, the SPI plus destination address, or SPI plus destination and source addresses are used to look up the SA. (See Section 4.1 for details on the algorithm that MUST be used for mapping inbound IPsec datagrams to SAs.) The following parameters are associated with each entry in

the SAD. They should all be present except where otherwise noted, e.g., AH Authentication algorithm. This description does not purport to be a MIB, only a specification of the minimal data items required to support an SA in an IPsec implementation.

For each of the selectors defined in Section 4.4.1.1, the entry for an inbound SA in the SAD MUST be initially populated with the value or values negotiated at the time the SA was created. (See the paragraph in Section 4.4.1 under "Handling Changes to the SPD while the System is Running" for guidance on the effect of SPD changes on extant SAs.) For a receiver, these values are used to check that the header fields of an inbound packet (after IPsec processing) match the selector values negotiated for the SA. Thus, the SAD acts as a cache for checking the selectors of inbound traffic arriving on SAs. For the receiver, this is part of verifying that a packet arriving on an SA is consistent with the policy for the SA. (See Section 6 for rules for ICMP messages.) These fields can have the form of specific values, ranges, ANY, or OPAQUE, as described in Section 4.4.1.1, "Selectors". Note also that there are a couple of situations in which the SAD can have entries for SAs that do not have corresponding entries in the SPD. Since this document does not mandate that the SAD be selectively cleared when the SPD is changed, SAD entries can remain when the SPD entries that created them are changed or deleted. Also, if a manually keyed SA is created, there could be an SAD entry for this SA that does not correspond to any SPD entry.

Note: The SAD can support multicast SAs, if manually configured. An outbound multicast SA has the same structure as a unicast SA. The source address is that of the sender, and the destination address is the multicast group address. An inbound, multicast SA must be configured with the source addresses of each peer authorized to transmit to the multicast SA in question. The SPI value for a multicast SA is provided by a multicast group controller, not by the receiver, as for a unicast SA. Because an SAD entry may be required to accommodate multiple, individual IP source addresses that were part of an SPD entry (for unicast SAs), the required facility for inbound, multicast SAs is a feature already present in an IPsec implementation. However, because the SPD has no provisions for accommodating multicast entries, this document does not specify an automated way to create an SAD entry for a multicast, inbound SA. Only manually configured SAD entries can be created to accommodate inbound, multicast traffic.

Implementation Guidance: This document does not specify how an SPD-S entry refers to the corresponding SAD entry, as this is an implementation-specific detail. However, some implementations (based on experience from RFC 2401) are known to have problems in this regard. In particular, simply storing the (remote tunnel header IP

address, remote SPI) pair in the SPD cache is not sufficient, since the pair does not always uniquely identify a single SAD entry. For instance, two hosts behind the same NAT could choose the same SPI value. The situation also may arise if a host is assigned an IP address (e.g., via DHCP) previously used by some other host, and the SAs associated with the old host have not yet been deleted via dead peer detection mechanisms. This may lead to packets being sent over the wrong SA or, if key management ensures the pair is unique, denying the creation of otherwise valid SAs. Thus, implementors should implement links between the SPD cache and the SAD in a way that does not engender such problems.

4.4.2.1. Data Items in the SAD

The following data items MUST be in the SAD:

- Security Parameter Index (SPI): a 32-bit value selected by the receiving end of an SA to uniquely identify the SA. In an SAD entry for an outbound SA, the SPI is used to construct the packet's AH or ESP header. In an SAD entry for an inbound SA, the SPI is used to map traffic to the appropriate SA (see text on unicast/multicast in Section 4.1).

- Sequence Number Counter: a 64-bit counter used to generate the Sequence Number field in AH or ESP headers. 64-bit sequence numbers are the default, but 32-bit sequence numbers are also supported if negotiated.

- Sequence Counter Overflow: a flag indicating whether overflow of the sequence number counter should generate an auditable event and prevent transmission of additional packets on the SA, or whether rollover is permitted. The audit log entry for this event SHOULD include the SPI value, current date/time, Local Address, Remote Address, and the selectors from the relevant SAD entry.

- Anti-Replay Window: a 64-bit counter and a bit-map (or equivalent) used to determine whether an inbound AH or ESP packet is a replay.

  Note: If anti-replay has been disabled by the receiver for an SA, e.g., in the case of a manually keyed SA, then the Anti-Replay Window is ignored for the SA in question. 64-bit sequence numbers are the default, but this counter size accommodates 32-bit sequence numbers as well.

- AH Authentication algorithm, key, etc. This is required only if AH is supported.

- o ESP Encryption algorithm, key, mode, IV, etc. If a combined mode algorithm is used, these fields will not be applicable.

- o ESP integrity algorithm, keys, etc. If the integrity service is not selected, these fields will not be applicable. If a combined mode algorithm is used, these fields will not be applicable.

- o ESP combined mode algorithms, key(s), etc. This data is used when a combined mode (encryption and integrity) algorithm is used with ESP. If a combined mode algorithm is not used, these fields are not applicable.

- o Lifetime of this SA: a time interval after which an SA must be replaced with a new SA (and new SPI) or terminated, plus an indication of which of these actions should occur. This may be expressed as a time or byte count, or a simultaneous use of both with the first lifetime to expire taking precedence. A compliant implementation MUST support both types of lifetimes, and MUST support a simultaneous use of both. If time is employed, and if IKE employs X.509 certificates for SA establishment, the SA lifetime must be constrained by the validity intervals of the certificates, and the NextIssueDate of the Certificate Revocation Lists (CRLs) used in the IKE exchange for the SA. Both initiator and responder are responsible for constraining the SA lifetime in this fashion. Note: The details of how to handle the refreshing of keys when SAs expire is a local matter. However, one reasonable approach is:

  (a) If byte count is used, then the implementation SHOULD count the number of bytes to which the IPsec cryptographic algorithm is applied. For ESP, this is the encryption algorithm (including Null encryption) and for AH, this is the authentication algorithm. This includes pad bytes, etc. Note that implementations MUST be able to handle having the counters at the ends of an SA get out of synch, e.g., because of packet loss or because the implementations at each end of the SA aren’t doing things the same way.

  (b) There SHOULD be two kinds of lifetime -- a soft lifetime that warns the implementation to initiate action such as setting up a replacement SA, and a hard lifetime when the current SA ends and is destroyed.

  (c) If the entire packet does not get delivered during the SA’s lifetime, the packet SHOULD be discarded.

- o IPsec protocol mode: tunnel or transport. Indicates which mode of AH or ESP is applied to traffic on this SA.

- o Stateful fragment checking flag. Indicates whether or not stateful fragment checking applies to this SA.

- o Bypass DF bit (T/F) -- applicable to tunnel mode SAs where both inner and outer headers are IPv4.

- o DSCP values -- the set of DSCP values allowed for packets carried over this SA. If no values are specified, no DSCP-specific filtering is applied. If one or more values are specified, these are used to select one SA among several that match the traffic selectors for an outbound packet. Note that these values are NOT checked against inbound traffic arriving on the SA.

- o Bypass DSCP (T/F) or map to unprotected DSCP values (array) if needed to restrict bypass of DSCP values -- applicable to tunnel mode SAs. This feature maps DSCP values from an inner header to values in an outer header, e.g., to address covert channel signaling concerns.

- o Path MTU: any observed path MTU and aging variables.

- o Tunnel header IP source and destination address -- both addresses must be either IPv4 or IPv6 addresses. The version implies the type of IP header to be used. Only used when the IPsec protocol mode is tunnel.

4.4.2.2. Relationship between SPD, PFP flag, packet, and SAD

For each selector, the following tables show the relationship between the value in the SPD, the PFP flag, the value in the triggering packet, and the resulting value in the SAD. Note that the administrative interface for IPsec can use various syntactic options to make it easier for the administrator to enter rules. For example, although a list of ranges is what IKEv2 sends, it might be clearer and less error prone for the user to enter a single IP address or IP address prefix.

Selector	SPD Entry	PFP	Value in Triggering Packet	Resulting SAD Entry
loc addr	list of ranges	0	IP addr "S"	list of ranges
	ANY	0	IP addr "S"	ANY
	list of ranges	1	IP addr "S"	"S"
	ANY	1	IP addr "S"	"S"
rem addr	list of ranges	0	IP addr "D"	list of ranges
	ANY	0	IP addr "D"	ANY
	list of ranges	1	IP addr "D"	"D"
	ANY	1	IP addr "D"	"D"
protocol	list of prot's*	0	prot. "P"	list of prot's*
	ANY**	0	prot. "P"	ANY
	OPAQUE****	0	prot. "P"	OPAQUE
	list of prot's*	0	not avail.	discard packet
	ANY**	0	not avail.	ANY
	OPAQUE****	0	not avail.	OPAQUE
	list of prot's*	1	prot. "P"	"P"
	ANY**	1	prot. "P"	"P"
	OPAQUE****	1	prot. "P"	***
	list of prot's*	1	not avail.	discard packet
	ANY**	1	not avail.	discard packet
	OPAQUE****	1	not avail.	***

If the protocol is one that has two ports, then there will be selectors for both Local and Remote ports.

Selector	SPD Entry	PFP	Value in Triggering Packet	Resulting SAD Entry
loc port	list of ranges	0	src port "s"	list of ranges
	ANY	0	src port "s"	ANY
	OPAQUE	0	src port "s"	OPAQUE
	list of ranges	0	not avail.	discard packet
	ANY	0	not avail.	ANY
	OPAQUE	0	not avail.	OPAQUE
	list of ranges	1	src port "s"	"s"
	ANY	1	src port "s"	"s"
	OPAQUE	1	src port "s"	***
	list of ranges	1	not avail.	discard packet
	ANY	1	not avail.	discard packet
	OPAQUE	1	not avail.	***
rem port	list of ranges	0	dst port "d"	list of ranges
	ANY	0	dst port "d"	ANY
	OPAQUE	0	dst port "d"	OPAQUE
	list of ranges	0	not avail.	discard packet
	ANY	0	not avail.	ANY
	OPAQUE	0	not avail.	OPAQUE
	list of ranges	1	dst port "d"	"d"
	ANY	1	dst port "d"	"d"
	OPAQUE	1	dst port "d"	***
	list of ranges	1	not avail.	discard packet
	ANY	1	not avail.	discard packet
	OPAQUE	1	not avail.	***

If the protocol is mobility header, then there will be a selector for mh type.

Selector	SPD Entry	PFP	Value in Triggering Packet	Resulting SAD Entry
mh type	list of ranges	0	mh type "T"	list of ranges
	ANY	0	mh type "T"	ANY
	OPAQUE	0	mh type "T"	OPAQUE
	list of ranges	0	not avail.	discard packet
	ANY	0	not avail.	ANY
	OPAQUE	0	not avail.	OPAQUE
	list of ranges	1	mh type "T"	"T"
	ANY	1	mh type "T"	"T"
	OPAQUE	1	mh type "T"	***
	list of ranges	1	not avail.	discard packet
	ANY	1	not avail.	discard packet
	OPAQUE	1	not avail.	***

If the protocol is ICMP, then there will be a 16-bit selector for ICMP type and ICMP code. Note that the type and code are bound to each other, i.e., the codes apply to the particular type. This 16-bit selector can contain a single type and a range of codes, a single type and ANY code, and ANY type and ANY code.

Selector	SPD Entry	PFP	Value in Triggering Packet	Resulting SAD Entry
ICMP type and code	a single type & range of codes	0	type "t" & code "c"	single type & range of codes
	a single type & ANY code	0	type "t" & code "c"	single type & ANY code
	ANY type & ANY code	0	type "t" & code "c"	ANY type & ANY code
	OPAQUE	0	type "t" & code "c"	OPAQUE
	a single type & range of codes	0	not avail.	discard packet
	a single type & ANY code	0	not avail.	discard packet
	ANY type & ANY code	0	not avail.	ANY type & ANY code
	OPAQUE	0	not avail.	OPAQUE
	a single type & range of codes	1	type "t" & code "c"	"t" and "c"
	a single type & ANY code	1	type "t" & code "c"	"t" and "c"
	ANY type & ANY code	1	type "t" & code "c"	"t" and "c"
	OPAQUE	1	type "t" & code "c"	***
	a single type & range of codes	1	not avail.	discard packet
	a single type & ANY code	1	not avail.	discard packet
	ANY type & ANY code	1	not avail.	discard packet
	OPAQUE	1	not avail.	***

If the name selector is used:

Selector	SPD Entry	PFP	Value in Triggering Packet	Resulting SAD Entry
name	list of user or system names	N/A	N/A	N/A

* "List of protocols" is the information, not the way that the SPD or SAD or IKEv2 have to represent this information.
** 0 (zero) is used by IKE to indicate ANY for protocol.
*** Use of PFP=1 with an OPAQUE value is an error and SHOULD be prohibited by an IPsec implementation.
**** The protocol field cannot be OPAQUE in IPv4. This table entry applies only to IPv6.

4.4.3. Peer Authorization Database (PAD)

The Peer Authorization Database (PAD) provides the link between the SPD and a security association management protocol such as IKE. It embodies several critical functions:

- identifies the peers or groups of peers that are authorized to communicate with this IPsec entity
- specifies the protocol and method used to authenticate each peer
- provides the authentication data for each peer
- constrains the types and values of IDs that can be asserted by a peer with regard to child SA creation, to ensure that the peer does not assert identities for lookup in the SPD that it is not authorized to represent, when child SAs are created
- peer gateway location info, e.g., IP address(es) or DNS names, MAY be included for peers that are known to be "behind" a security gateway

The PAD provides these functions for an IKE peer when the peer acts as either the initiator or the responder.

To perform these functions, the PAD contains an entry for each peer or group of peers with which the IPsec entity will communicate. An entry names an individual peer (a user, end system or security gateway) or specifies a group of peers (using ID matching rules defined below). The entry specifies the authentication protocol (e.g., IKEv1, IKEv2, KINK) method used (e.g., certificates or pre-shared secrets) and the authentication data (e.g., the pre-shared

secret or the trust anchor relative to which the peer's certificate will be validated). For certificate-based authentication, the entry also may provide information to assist in verifying the revocation status of the peer, e.g., a pointer to a CRL repository or the name of an Online Certificate Status Protocol (OCSP) server associated with the peer or with the trust anchor associated with the peer.

Each entry also specifies whether the IKE ID payload will be used as a symbolic name for SPD lookup, or whether the remote IP address provided in traffic selector payloads will be used for SPD lookups when child SAs are created.

Note that the PAD information MAY be used to support creation of more than one tunnel mode SA at a time between two peers, e.g., two tunnels to protect the same addresses/hosts, but with different tunnel endpoints.

4.4.3.1. PAD Entry IDs and Matching Rules

The PAD is an ordered database, where the order is defined by an administrator (or a user in the case of a single-user end system). Usually, the same administrator will be responsible for both the PAD and SPD, since the two databases must be coordinated. The ordering requirement for the PAD arises for the same reason as for the SPD, i.e., because use of "star name" entries allows for overlaps in the set of IKE IDs that could match a specific entry.

Six types of IDs are supported for entries in the PAD, consistent with the symbolic name types and IP addresses used to identify SPD entries. The ID for each entry acts as the index for the PAD, i.e., it is the value used to select an entry. All of these ID types can be used to match IKE ID payload types. The six types are:

- DNS name (specific or partial)
- Distinguished Name (complete or sub-tree constrained)
- RFC 822 email address (complete or partially qualified)
- IPv4 address (range)
- IPv6 address (range)
- Key ID (exact match only)

The first three name types can accommodate sub-tree matching as well as exact matches. A DNS name may be fully qualified and thus match exactly one name, e.g., foo.example.com. Alternatively, the name may encompass a group of peers by being partially specified, e.g., the string ".example.com" could be used to match any DNS name ending in these two domain name components.

Similarly, a Distinguished Name may specify a complete Distinguished Name to match exactly one entry, e.g., CN = Stephen, O = BBN Technologies, SP = MA, C = US. Alternatively, an entry may encompass a group of peers by specifying a sub-tree, e.g., an entry of the form "C = US, SP = MA" might be used to match all DNs that contain these two attributes as the top two Relative Distinguished Names (RDNs).

For an RFC 822 e-mail addresses, the same options exist. A complete address such as foo@example.com matches one entity, but a sub-tree name such as "@example.com" could be used to match all the entities with names ending in those two domain names to the right of the @.

The specific syntax used by an implementation to accommodate sub-tree matching for distinguished names, domain names or RFC 822 e-mail addresses is a local matter. But, at a minimum, sub-tree matching of the sort described above MUST be supported. (Substring matching within a DN, DNS name, or RFC 822 address MAY be supported, but is not required.)

For IPv4 and IPv6 addresses, the same address range syntax used for SPD entries MUST be supported. This allows specification of an individual address (via a trivial range), an address prefix (by choosing a range that adheres to Classless Inter-Domain Routing (CIDR)-style prefixes), or an arbitrary address range.

The Key ID field is defined as an OCTET string in IKE. For this name type, only exact-match syntax MUST be supported (since there is no explicit structure for this ID type). Additional matching functions MAY be supported for this ID type.

4.4.3.2. IKE Peer Authentication Data

Once an entry is located based on an ordered search of the PAD based on ID field matching, it is necessary to verify the asserted identity, i.e., to authenticate the asserted ID. For each PAD entry, there is an indication of the type of authentication to be performed. This document requires support for two required authentication data types:

- X.509 certificate
- pre-shared secret

For authentication based on an X.509 certificate, the PAD entry contains a trust anchor via which the end entity (EE) certificate for the peer must be verifiable, either directly or via a certificate path. See RFC 3280 for the definition of a trust anchor. An entry used with certificate-based authentication MAY include additional data to facilitate certificate revocation status, e.g., a list of

appropriate OCSP responders or CRL repositories, and associated authentication data. For authentication based on a pre-shared secret, the PAD contains the pre-shared secret to be used by IKE.

This document does not require that the IKE ID asserted by a peer be syntactically related to a specific field in an end entity certificate that is employed to authenticate the identity of that peer. However, it often will be appropriate to impose such a requirement, e.g., when a single entry represents a set of peers each of whom may have a distinct SPD entry. Thus, implementations MUST provide a means for an administrator to require a match between an asserted IKE ID and the subject name or subject alt name in a certificate. The former is applicable to IKE IDs expressed as distinguished names; the latter is appropriate for DNS names, RFC 822 e-mail addresses, and IP addresses. Since KEY ID is intended for identifying a peer authenticated via a pre-shared secret, there is no requirement to match this ID type to a certificate field.

See IKEv1 [HarCar98] and IKEv2 [Kau05] for details of how IKE performs peer authentication using certificates or pre-shared secrets.

This document does not mandate support for any other authentication methods, although such methods MAY be employed.

4.4.3.3. Child SA Authorization Data

Once an IKE peer is authenticated, child SAs may be created. Each PAD entry contains data to constrain the set of IDs that can be asserted by an IKE peer, for matching against the SPD. Each PAD entry indicates whether the IKE ID is to be used as a symbolic name for SPD matching, or whether an IP address asserted in a traffic selector payload is to be used.

If the entry indicates that the IKE ID is to be used, then the PAD entry ID field defines the authorized set of IDs. If the entry indicates that child SAs traffic selectors are to be used, then an additional data element is required, in the form of IPv4 and/or IPv6 address ranges. (A peer may be authorized for both address types, so there MUST be provision for both a v4 and a v6 address range.)

4.4.3.4. How the PAD Is Used

During the initial IKE exchange, the initiator and responder each assert their identity via the IKE ID payload and send an AUTH payload to verify the asserted identity. One or more CERT payloads may be transmitted to facilitate the verification of each asserted identity.

When an IKE entity receives an IKE ID payload, it uses the asserted ID to locate an entry in the PAD, using the matching rules described above. The PAD entry specifies the authentication method to be employed for the identified peer. This ensures that the right method is used for each peer and that different methods can be used for different peers. The entry also specifies the authentication data that will be used to verify the asserted identity. This data is employed in conjunction with the specified method to authenticate the peer, before any CHILD SAs are created.

Child SAs are created based on the exchange of traffic selector payloads, either at the end of the initial IKE exchange or in subsequent CREATE_CHILD_SA exchanges. The PAD entry for the (now authenticated) IKE peer is used to constrain creation of child SAs; specifically, the PAD entry specifies how the SPD is searched using a traffic selector proposal from a peer. There are two choices: either the IKE ID asserted by the peer is used to find an SPD entry via its symbolic name, or peer IP addresses asserted in traffic selector payloads are used for SPD lookups based on the remote IP address field portion of an SPD entry. It is necessary to impose these constraints on creation of child SAs to prevent an authenticated peer from spoofing IDs associated with other, legitimate peers.

Note that because the PAD is checked before searching for an SPD entry, this safeguard protects an initiator against spoofing attacks. For example, assume that IKE A receives an outbound packet destined for IP address X, a host served by a security gateway. RFC 2401 [RFC2401] and this document do not specify how A determines the address of the IKE peer serving X. However, any peer contacted by A as the presumed representative for X must be registered in the PAD in order to allow the IKE exchange to be authenticated. Moreover, when the authenticated peer asserts that it represents X in its traffic selector exchange, the PAD will be consulted to determine if the peer in question is authorized to represent X. Thus, the PAD provides a binding of address ranges (or name sub-spaces) to peers, to counter such attacks.

4.5. SA and Key Management

All IPsec implementations MUST support both manual and automated SA and cryptographic key management. The IPsec protocols, AH and ESP, are largely independent of the associated SA management techniques, although the techniques involved do affect some of the security services offered by the protocols. For example, the optional anti-replay service available for AH and ESP requires automated SA management. Moreover, the granularity of key distribution employed with IPsec determines the granularity of authentication provided. In general, data origin authentication in AH and ESP is limited by the

extent to which secrets used with the integrity algorithm (or with a key management protocol that creates such secrets) are shared among multiple possible sources.

The following text describes the minimum requirements for both types of SA management.

4.5.1. Manual Techniques

The simplest form of management is manual management, in which a person manually configures each system with keying material and SA management data relevant to secure communication with other systems. Manual techniques are practical in small, static environments but they do not scale well. For example, a company could create a virtual private network (VPN) using IPsec in security gateways at several sites. If the number of sites is small, and since all the sites come under the purview of a single administrative domain, this might be a feasible context for manual management techniques. In this case, the security gateway might selectively protect traffic to and from other sites within the organization using a manually configured key, while not protecting traffic for other destinations. It also might be appropriate when only selected communications need to be secured. A similar argument might apply to use of IPsec entirely within an organization for a small number of hosts and/or gateways. Manual management techniques often employ statically configured, symmetric keys, though other options also exist.

4.5.2. Automated SA and Key Management

Widespread deployment and use of IPsec requires an Internet-standard, scalable, automated, SA management protocol. Such support is required to facilitate use of the anti-replay features of AH and ESP, and to accommodate on-demand creation of SAs, e.g., for user- and session-oriented keying. (Note that the notion of "rekeying" an SA actually implies creation of a new SA with a new SPI, a process that generally implies use of an automated SA/key management protocol.)

The default automated key management protocol selected for use with IPsec is IKEv2 [Kau05]. This document assumes the availability of certain functions from the key management protocol that are not supported by IKEv1. Other automated SA management protocols MAY be employed.

When an automated SA/key management protocol is employed, the output from this protocol is used to generate multiple keys for a single SA. This also occurs because distinct keys are used for each of the two

SAs created by IKE. If both integrity and confidentiality are employed, then a minimum of four keys are required. Additionally, some cryptographic algorithms may require multiple keys, e.g., 3DES.

The Key Management System may provide a separate string of bits for each key or it may generate one string of bits from which all keys are extracted. If a single string of bits is provided, care needs to be taken to ensure that the parts of the system that map the string of bits to the required keys do so in the same fashion at both ends of the SA. To ensure that the IPsec implementations at each end of the SA use the same bits for the same keys, and irrespective of which part of the system divides the string of bits into individual keys, the encryption keys MUST be taken from the first (left-most, high-order) bits and the integrity keys MUST be taken from the remaining bits. The number of bits for each key is defined in the relevant cryptographic algorithm specification RFC. In the case of multiple encryption keys or multiple integrity keys, the specification for the cryptographic algorithm must specify the order in which they are to be selected from a single string of bits provided to the cryptographic algorithm.

4.5.3. Locating a Security Gateway

This section discusses issues relating to how a host learns about the existence of relevant security gateways and, once a host has contacted these security gateways, how it knows that these are the correct security gateways. The details of where the required information is stored is a local matter, but the Peer Authorization Database (PAD) described in Section 4.4 is the most likely candidate. (Note: S* indicates a system that is running IPsec, e.g., SH1 and SG2 below.)

Consider a situation in which a remote host (SH1) is using the Internet to gain access to a server or other machine (H2) and there is a security gateway (SG2), e.g., a firewall, through which H1's traffic must pass. An example of this situation would be a mobile host crossing the Internet to his home organization's firewall (SG2). This situation raises several issues:

1. How does SH1 know/learn about the existence of the security gateway SG2?

2. How does it authenticate SG2, and once it has authenticated SG2, how does it confirm that SG2 has been authorized to represent H2?

3. How does SG2 authenticate SH1 and verify that SH1 is authorized to contact H2?

4. How does SH1 know/learn about any additional gateways that provide alternate paths to H2?

To address these problems, an IPsec-supporting host or security gateway MUST have an administrative interface that allows the user/administrator to configure the address of one or more security gateways for ranges of destination addresses that require its use. This includes the ability to configure information for locating and authenticating one or more security gateways and verifying the authorization of these gateways to represent the destination host. (The authorization function is implied in the PAD.) This document does not address the issue of how to automate the discovery/verification of security gateways.

4.6. SAs and Multicast

The receiver-orientation of the SA implies that, in the case of unicast traffic, the destination system will select the SPI value. By having the destination select the SPI value, there is no potential for manually configured SAs to conflict with automatically configured (e.g., via a key management protocol) SAs or for SAs from multiple sources to conflict with each other. For multicast traffic, there are multiple destination systems associated with a single SA. So some system or person will need to coordinate among all multicast groups to select an SPI or SPIs on behalf of each multicast group and then communicate the group's IPsec information to all of the legitimate members of that multicast group via mechanisms not defined here.

Multiple senders to a multicast group SHOULD use a single Security Association (and hence SPI) for all traffic to that group when a symmetric key encryption or integrity algorithm is employed. In such circumstances, the receiver knows only that the message came from a system possessing the key for that multicast group. In such circumstances, a receiver generally will not be able to authenticate which system sent the multicast traffic. Specifications for other, more general multicast approaches are deferred to the IETF Multicast Security Working Group.

5. IP Traffic Processing

As mentioned in Section 4.4.1, "The Security Policy Database (SPD)", the SPD (or associated caches) MUST be consulted during the processing of all traffic that crosses the IPsec protection boundary, including IPsec management traffic. If no policy is found in the SPD that matches a packet (for either inbound or outbound traffic), the packet MUST be discarded. To simplify processing, and to allow for very fast SA lookups (for SG/BITS/BITW), this document introduces the

notion of an SPD cache for all outbound traffic (SPD-O plus SPD-S), and a cache for inbound, non-IPsec-protected traffic (SPD-I). (As mentioned earlier, the SAD acts as a cache for checking the selectors of inbound IPsec-protected traffic arriving on SAs.) There is nominally one cache per SPD. For the purposes of this specification, it is assumed that each cached entry will map to exactly one SA. Note, however, exceptions arise when one uses multiple SAs to carry traffic of different priorities (e.g., as indicated by distinct DSCP values) but the same selectors. Note also, that there are a couple of situations in which the SAD can have entries for SAs that do not have corresponding entries in the SPD. Since this document does not mandate that the SAD be selectively cleared when the SPD is changed, SAD entries can remain when the SPD entries that created them are changed or deleted. Also, if a manually keyed SA is created, there could be an SAD entry for this SA that does not correspond to any SPD entry.

Since SPD entries may overlap, one cannot safely cache these entries in general. Simple caching might result in a match against a cache entry, whereas an ordered search of the SPD would have resulted in a match against a different entry. But, if the SPD entries are first decorrelated, then the resulting entries can safely be cached. Each cached entry will indicate that matching traffic should be bypassed or discarded, appropriately. (Note: The original SPD entry might result in multiple SAs, e.g., because of PFP.) Unless otherwise noted, all references below to the "SPD" or "SPD cache" or "cache" are to a decorrelated SPD (SPD-I, SPD-O, SPD-S) or the SPD cache containing entries from the decorrelated SPD.

Note: In a host IPsec implementation based on sockets, the SPD will be consulted whenever a new socket is created to determine what, if any, IPsec processing will be applied to the traffic that will flow on that socket. This provides an implicit caching mechanism, and the portions of the preceding discussion that address caching can be ignored in such implementations.

Note: It is assumed that one starts with a correlated SPD because that is how users and administrators are accustomed to managing these sorts of access control lists or firewall filter rules. Then the decorrelation algorithm is applied to build a list of cache-able SPD entries. The decorrelation is invisible at the management interface.

For inbound IPsec traffic, the SAD entry selected by the SPI serves as the cache for the selectors to be matched against arriving IPsec packets, after AH or ESP processing has been performed.

5.1. Outbound IP Traffic Processing (protected-to-unprotected)

First consider the path for traffic entering the implementation via a protected interface and exiting via an unprotected interface.

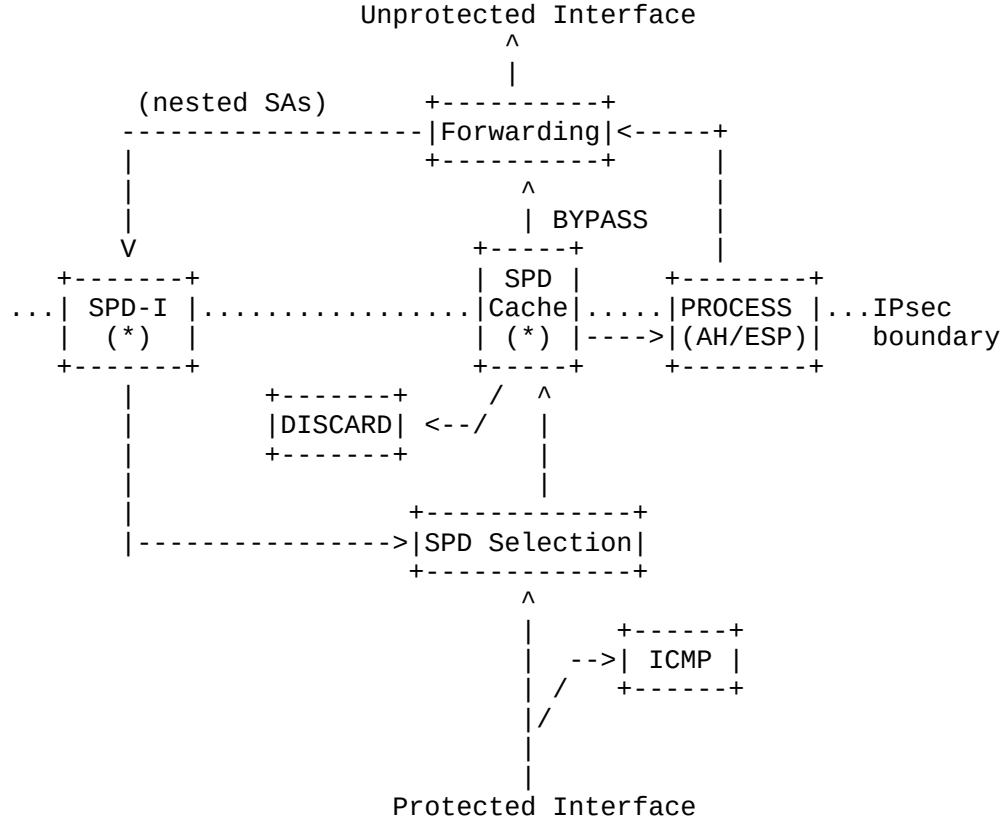


Figure 2. Processing Model for Outbound Traffic
(*) = The SPD caches are shown here. If there is a cache miss, then the SPD is checked. There is no requirement that an implementation buffer the packet if there is a cache miss.

IPsec MUST perform the following steps when processing outbound packets:

1. When a packet arrives from the subscriber (protected) interface, invoke the SPD selection function to obtain the SPD-ID needed to choose the appropriate SPD. (If the implementation uses only one SPD, this step is a no-op.)

2. Match the packet headers against the cache for the SPD specified by the SPD-ID from step 1. Note that this cache contains entries from SPD-O and SPD-S.

3a. If there is a match, then process the packet as specified by the matching cache entry, i.e., BYPASS, DISCARD, or PROTECT using AH or ESP. If IPsec processing is applied, there is a link from the SPD cache entry to the relevant SAD entry (specifying the mode, cryptographic algorithms, keys, SPI, PMTU, etc.). IPsec processing is as previously defined, for tunnel or transport modes and for AH or ESP, as specified in their respective RFCs [Ken05b, Ken05a]. Note that the SA PMTU value, plus the value of the stateful fragment checking flag (and the DF bit in the IP header of the outbound packet) determine whether the packet can (must) be fragmented prior to or after IPsec processing, or if it must be discarded and an ICMP PMTU message is sent.

3b. If no match is found in the cache, search the SPD (SPD-S and SPD-O parts) specified by SPD-ID. If the SPD entry calls for BYPASS or DISCARD, create one or more new outbound SPD cache entries and if BYPASS, create one or more new inbound SPD cache entries. (More than one cache entry may be created since a decorrelated SPD entry may be linked to other such entries that were created as a side effect of the decorrelation process.) If the SPD entry calls for PROTECT, i.e., creation of an SA, the key management mechanism (e.g., IKEv2) is invoked to create the SA. If SA creation succeeds, a new outbound (SPD-S) cache entry is created, along with outbound and inbound SAD entries, otherwise the packet is discarded. (A packet that triggers an SPD lookup MAY be discarded by the implementation, or it MAY be processed against the newly created cache entry, if one is created.) Since SAs are created in pairs, an SAD entry for the corresponding inbound SA also is created, and it contains the selector values derived from the SPD entry (and packet, if any PFP flags were "true") used to create the inbound SA, for use in checking inbound traffic delivered via the SA.

4. The packet is passed to the outbound forwarding function (operating outside of the IPsec implementation), to select the interface to which the packet will be directed. This function

may cause the packet to be passed back across the IPsec boundary, for additional IPsec processing, e.g., in support of nested SAs. If so, there MUST be an entry in SPD-I database that permits inbound bypassing of the packet, otherwise the packet will be discarded. If necessary, i.e., if there is more than one SPD-I, the traffic being looped back MAY be tagged as coming from this internal interface. This would allow the use of a different SPD-I for "real" external traffic vs. looped traffic, if needed.

Note: With the exception of IPv4 and IPv6 transport mode, an SG, BITS, or BITW implementation MAY fragment packets before applying IPsec. (This applies only to IPv4. For IPv6 packets, only the originator is allowed to fragment them.) The device SHOULD have a configuration setting to disable this. The resulting fragments are evaluated against the SPD in the normal manner. Thus, fragments not containing port numbers (or ICMP message type and code, or Mobility Header type) will only match rules having port (or ICMP message type and code, or MH type) selectors of OPAQUE or ANY. (See Section 7 for more details.)

Note: With regard to determining and enforcing the PMTU of an SA, the IPsec system MUST follow the steps described in Section 8.2.

5.1.1. Handling an Outbound Packet That Must Be Discarded

If an IPsec system receives an outbound packet that it finds it must discard, it SHOULD be capable of generating and sending an ICMP message to indicate to the sender of the outbound packet that the packet was discarded. The type and code of the ICMP message will depend on the reason for discarding the packet, as specified below. The reason SHOULD be recorded in the audit log. The audit log entry for this event SHOULD include the reason, current date/time, and the selector values from the packet.

a. The selectors of the packet matched an SPD entry requiring the packet to be discarded.

   IPv4 Type = 3 (destination unreachable) Code = 13
   (Communication Administratively Prohibited)

   IPv6 Type = 1 (destination unreachable) Code = 1
   (Communication with destination administratively prohibited)

b1. The IPsec system successfully reached the remote peer but was unable to negotiate the SA required by the SPD entry matching the packet because, for example, the remote peer is administratively prohibited from communicating with the initiator, the initiating

peer was unable to authenticate itself to the remote peer, the remote peer was unable to authenticate itself to the initiating peer, or the SPD at the remote peer did not have a suitable entry.

IPv4 Type = 3 (destination unreachable) Code = 13 (Communication Administratively Prohibited)

IPv6 Type = 1 (destination unreachable) Code = 1 (Communication with destination administratively prohibited)

b2. The IPsec system was unable to set up the SA required by the SPD entry matching the packet because the IPsec peer at the other end of the exchange could not be contacted.

IPv4 Type = 3 (destination unreachable) Code = 1 (host unreachable)

IPv6 Type = 1 (destination unreachable) Code = 3 (address unreachable)

Note that an attacker behind a security gateway could send packets with a spoofed source address, W.X.Y.Z, to an IPsec entity causing it to send ICMP messages to W.X.Y.Z. This creates an opportunity for a denial of service (DoS) attack among hosts behind a security gateway. To address this, a security gateway SHOULD include a management control to allow an administrator to configure an IPsec implementation to send or not send the ICMP messages under these circumstances, and if this facility is selected, to rate limit the transmission of such ICMP responses.

5.1.2. Header Construction for Tunnel Mode

This section describes the handling of the inner and outer IP headers, extension headers, and options for AH and ESP tunnels, with regard to outbound traffic processing. This includes how to construct the encapsulating (outer) IP header, how to process fields in the inner IP header, and what other actions should be taken for outbound, tunnel mode traffic. The general processing described here is modeled after RFC 2003, "IP Encapsulation within IP" [Per96]:

- The outer IP header Source Address and Destination Address identify the "endpoints" of the tunnel (the encapsulator and decapsulator). The inner IP header Source Address and Destination Addresses identify the original sender and recipient of the datagram (from the perspective of this tunnel), respectively.

(See footnote 3 after the table in 5.1.2.1 for more details on the encapsulating source IP address.)

- o The inner IP header is not changed except as noted below for TTL (or Hop Limit) and the DS/ECN Fields. The inner IP header otherwise remains unchanged during its delivery to the tunnel exit point.

- o No change to IP options or extension headers in the inner header occurs during delivery of the encapsulated datagram through the tunnel.

Note: IPsec tunnel mode is different from IP-in-IP tunneling (RFC 2003 [Per96]) in several ways:

- o IPsec offers certain controls to a security administrator to manage covert channels (which would not normally be a concern for tunneling) and to ensure that the receiver examines the right portions of the received packet with respect to application of access controls. An IPsec implementation MAY be configurable with regard to how it processes the outer DS field for tunnel mode for transmitted packets. For outbound traffic, one configuration setting for the outer DS field will operate as described in the following sections on IPv4 and IPv6 header processing for IPsec tunnels. Another will allow the outer DS field to be mapped to a fixed value, which MAY be configured on a per-SA basis. (The value might really be fixed for all traffic outbound from a device, but per-SA granularity allows that as well.) This configuration option allows a local administrator to decide whether the covert channel provided by copying these bits outweighs the benefits of copying.

- o IPsec describes how to handle ECN or DS and provides the ability to control propagation of changes in these fields between unprotected and protected domains. In general, propagation from a protected to an unprotected domain is a covert channel and thus controls are provided to manage the bandwidth of this channel. Propagation of ECN values in the other direction are controlled so that only legitimate ECN changes (indicating occurrence of congestion between the tunnel endpoints) are propagated. By default, DS propagation from an unprotected domain to a protected domain is not permitted. However, if the sender and receiver do not share the same DS code space, and the receiver has no way of learning how to map between the two spaces, then it may be appropriate to deviate from the default. Specifically, an IPsec implementation MAY be configurable in terms of how it processes the outer DS field for tunnel mode for received packets. It may be configured to either discard the outer DS value (the default) OR to overwrite the inner DS field with the outer DS field. If

offered, the discard vs. overwrite behavior MAY be configured on a per-SA basis. This configuration option allows a local administrator to decide whether the vulnerabilities created by copying these bits outweigh the benefits of copying. See [RFC2983] for further information on when each of these behaviors may be useful, and also for the possible need for diffserv traffic conditioning prior or subsequent to IPsec processing (including tunnel decapsulation).

- o IPsec allows the IP version of the encapsulating header to be different from that of the inner header.

The tables in the following sub-sections show the handling for the different header/option fields ("constructed" means that the value in the outer field is constructed independently of the value in the inner).

5.1.2.1. IPv4: Header Construction for Tunnel Mode

	<-- How Outer Hdr Relates to	Inner Hdr -->
	Outer Hdr at	Inner Hdr at
IPv4	Encapsulator	Decapsulator
Header fields:	---	---
version	4 (1)	no change
header length	constructed	no change
DS Field	copied from inner hdr (5)	no change
ECN Field	copied from inner hdr	constructed (6)
total length	constructed	no change
ID	constructed	no change
flags (DF,MF)	constructed, DF (4)	no change
fragment offset	constructed	no change
TTL	constructed (2)	decrement (2)
protocol	AH, ESP	no change
checksum	constructed	constructed (2)(6)
src address	constructed (3)	no change
dest address	constructed (3)	no change
Options	never copied	no change

Notes:

(1) The IP version in the encapsulating header can be different from the value in the inner header.

(2) The TTL in the inner header is decremented by the encapsulator prior to forwarding and by the decapsulator if it forwards the packet. (The IPv4 checksum changes when the TTL changes.)

Note: Decrementing the TTL value is a normal part of forwarding a packet. Thus, a packet originating from the same node as the encapsulator does not have its TTL decremented, since the sending node is originating the packet rather than forwarding it. This applies to BITS and native IPsec implementations in hosts and routers. However, the IPsec processing model includes an external forwarding capability. TTL processing can be used to prevent looping of packets, e.g., due to configuration errors, within the context of this processing model.

(3) Local and Remote addresses depend on the SA, which is used to determine the Remote address, which in turn determines which Local address (net interface) is used to forward the packet.

Note: For multicast traffic, the destination address, or source and destination addresses, may be required for demuxing. In that case, it is important to ensure consistency over the lifetime of the SA by ensuring that the source address that appears in the encapsulating tunnel header is the same as the one that was negotiated during the SA establishment process. There is an exception to this general rule, i.e., a mobile IPsec implementation will update its source address as it moves.

(4) Configuration determines whether to copy from the inner header (IPv4 only), clear, or set the DF.

(5) If the packet will immediately enter a domain for which the DSCP value in the outer header is not appropriate, that value MUST be mapped to an appropriate value for the domain [NiBlBaBL98]. See RFC 2475 [BBCDWW98] for further information.

(6) If the ECN field in the inner header is set to ECT(0) or ECT(1), where ECT is ECN-Capable Transport (ECT), and if the ECN field in the outer header is set to Congestion Experienced (CE), then set the ECN field in the inner header to CE; otherwise, make no change to the ECN field in the inner header. (The IPv4 checksum changes when the ECN changes.)

Note: IPsec does not copy the options from the inner header into the outer header, nor does IPsec construct the options in the outer header. However, post-IPsec code MAY insert/construct options for the outer header.

5.1.2.2. IPv6: Header Construction for Tunnel Mode

	<-- How Outer Hdr Relates	Inner Hdr --->
	Outer Hdr at	Inner Hdr at
IPv6	Encapsulator	Decapsulator
Header fields:	----------	----------
version	6 (1)	no change
DS Field	copied from inner hdr (5)	no change (9)
ECN Field	copied from inner hdr	constructed (6)
flow label	copied or configured (8)	no change
payload length	constructed	no change
next header	AH,ESP,routing hdr	no change
hop limit	constructed (2)	decrement (2)
src address	constructed (3)	no change
dest address	constructed (3)	no change
Extension headers	never copied (7)	no change

Notes:

(1) - (6) See Section 5.1.2.1.

(7) IPsec does not copy the extension headers from the inner packet into outer headers, nor does IPsec construct extension headers in the outer header. However, post-IPsec code MAY insert/construct extension headers for the outer header.

(8) See [RaCoCaDe04]. Copying is acceptable only for end systems, not SGs. If an SG copied flow labels from the inner header to the outer header, collisions might result.

(9) An implementation MAY choose to provide a facility to pass the DS value from the outer header to the inner header, on a per-SA basis, for received tunnel mode packets. The motivation for providing this feature is to accommodate situations in which the DS code space at the receiver is different from that of the sender and the receiver has no way of knowing how to translate from the sender’s space. There is a danger in copying this value from the outer header to the inner header, since it enables an attacker to modify the outer DSCP value in a fashion that may adversely affect other traffic at the receiver. Hence the default behavior for IPsec implementations is NOT to permit such copying.

5.2. Processing Inbound IP Traffic (unprotected-to-protected)

Inbound processing is somewhat different from outbound processing, because of the use of SPIs to map IPsec-protected traffic to SAs. The inbound SPD cache (SPD-I) is applied only to bypassed or

discarded traffic. If an arriving packet appears to be an IPsec fragment from an unprotected interface, reassembly is performed prior to IPsec processing. The intent for any SPD cache is that a packet that fails to match any entry is then referred to the corresponding SPD. Every SPD SHOULD have a nominal, final entry that catches anything that is otherwise unmatched, and discards it. This ensures that non-IPsec-protected traffic that arrives and does not match any SPD-I entry will be discarded.

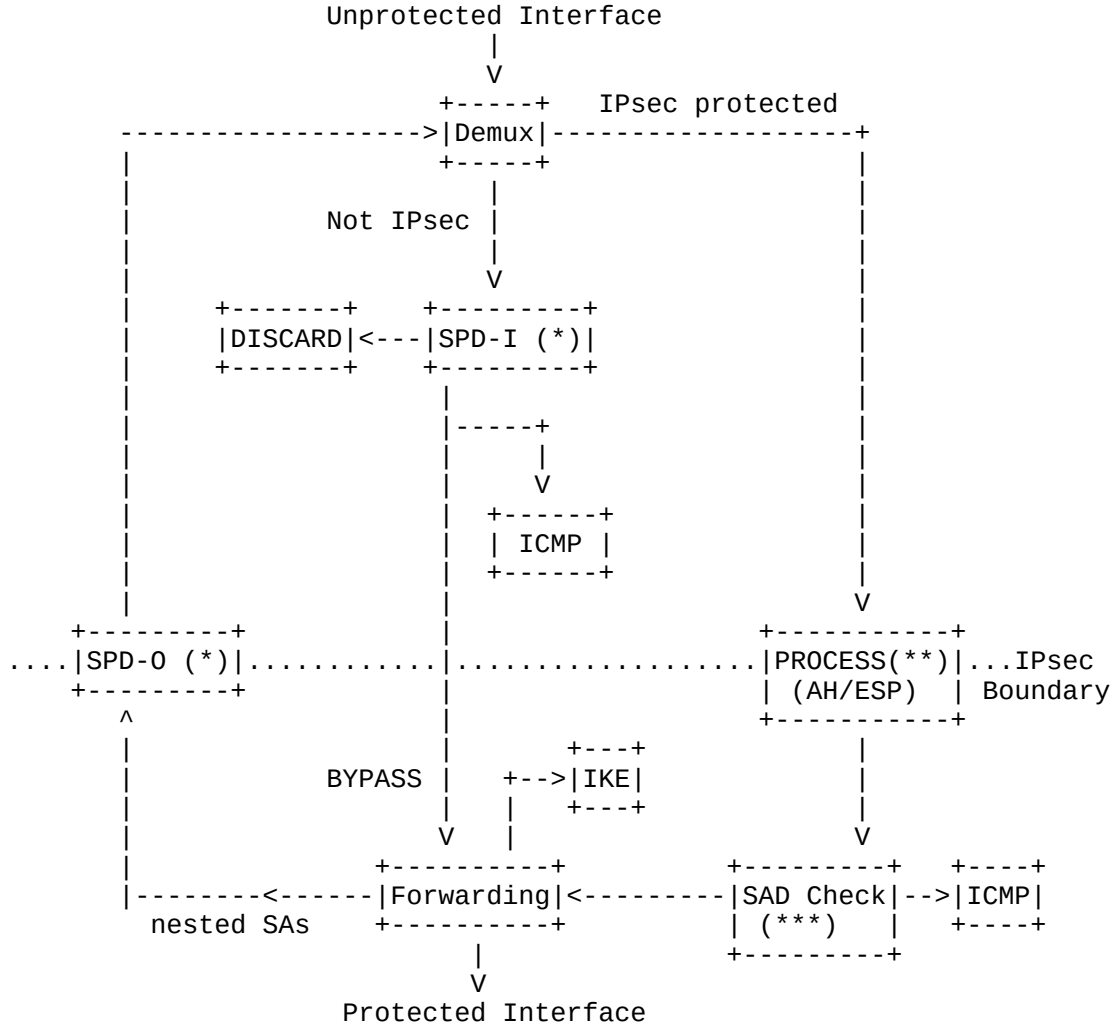


Figure 3. Processing Model for Inbound Traffic

(*) = The caches are shown here. If there is a cache miss, then the SPD is checked. There is no requirement that an implementation buffer the packet if there is a cache miss.
(**) = This processing includes using the packet's SPI, etc., to look up the SA in the SAD, which forms a cache of the SPD for inbound packets (except for cases noted in Sections 4.4.2 and 5). See step 3a below.
(***) = This SAD check refers to step 4 below.

Prior to performing AH or ESP processing, any IP fragments that arrive via the unprotected interface are reassembled (by IP). Each inbound IP datagram to which IPsec processing will be applied is identified by the appearance of the AH or ESP values in the IP Next Protocol field (or of AH or ESP as a next layer protocol in the IPv6 context).

IPsec MUST perform the following steps:

1. When a packet arrives, it may be tagged with the ID of the interface (physical or virtual) via which it arrived, if necessary, to support multiple SPDs and associated SPD-I caches. (The interface ID is mapped to a corresponding SPD-ID.)

2. The packet is examined and demuxed into one of two categories:
   - If the packet appears to be IPsec protected and it is addressed to this device, an attempt is made to map it to an active SA via the SAD. Note that the device may have multiple IP addresses that may be used in the SAD lookup, e.g., in the case of protocols such as SCTP.
   - Traffic not addressed to this device, or addressed to this device and not AH or ESP, is directed to SPD-I lookup. (This implies that IKE traffic MUST have an explicit BYPASS entry in the SPD.) If multiple SPDs are employed, the tag assigned to the packet in step 1 is used to select the appropriate SPD-I (and cache) to search. SPD-I lookup determines whether the action is DISCARD or BYPASS.

3a. If the packet is addressed to the IPsec device and AH or ESP is specified as the protocol, the packet is looked up in the SAD. For unicast traffic, use only the SPI (or SPI plus protocol). For multicast traffic, use the SPI plus the destination or SPI plus destination and source addresses, as specified in Section 4.1. In either case (unicast or multicast), if there is no match, discard the traffic. This is an auditable event. The audit log

entry for this event SHOULD include the current date/time, SPI, source and destination of the packet, IPsec protocol, and any other selector values of the packet that are available. If the packet is found in the SAD, process it accordingly (see step 4).

3b. If the packet is not addressed to the device or is addressed to this device and is not AH or ESP, look up the packet header in the (appropriate) SPD-I cache. If there is a match and the packet is to be discarded or bypassed, do so. If there is no cache match, look up the packet in the corresponding SPD-I and create a cache entry as appropriate. (No SAs are created in response to receipt of a packet that requires IPsec protection; only BYPASS or DISCARD cache entries can be created this way.) If there is no match, discard the traffic. This is an auditable event. The audit log entry for this event SHOULD include the current date/time, SPI if available, IPsec protocol if available, source and destination of the packet, and any other selector values of the packet that are available.

3c. Processing of ICMP messages is assumed to take place on the unprotected side of the IPsec boundary. Unprotected ICMP messages are examined and local policy is applied to determine whether to accept or reject these messages and, if accepted, what action to take as a result. For example, if an ICMP unreachable message is received, the implementation must decide whether to act on it, reject it, or act on it with constraints. (See Section 6.)

4. Apply AH or ESP processing as specified, using the SAD entry selected in step 3a above. Then match the packet against the inbound selectors identified by the SAD entry to verify that the received packet is appropriate for the SA via which it was received.

5. If an IPsec system receives an inbound packet on an SA and the packet's header fields are not consistent with the selectors for the SA, it MUST discard the packet. This is an auditable event. The audit log entry for this event SHOULD include the current date/time, SPI, IPsec protocol(s), source and destination of the packet, any other selector values of the packet that are available, and the selector values from the relevant SAD entry. The system SHOULD also be capable of generating and sending an IKE notification of INVALID_SELECTORS to the sender (IPsec peer), indicating that the received packet was discarded because of failure to pass selector checks.

To minimize the impact of a DoS attack, or a mis-configured peer, the IPsec system SHOULD include a management control to allow an administrator to configure the IPsec implementation to send or not send this IKE notification, and if this facility is selected, to rate limit the transmission of such notifications.

After traffic is bypassed or processed through IPsec, it is handed to the inbound forwarding function for disposition. This function may cause the packet to be sent (outbound) across the IPsec boundary for additional inbound IPsec processing, e.g., in support of nested SAs. If so, then as with ALL outbound traffic that is to be bypassed, the packet MUST be matched against an SPD-O entry. Ultimately, the packet should be forwarded to the destination host or process for disposition.

6. ICMP Processing

This section describes IPsec handling of ICMP traffic. There are two categories of ICMP traffic: error messages (e.g., type = destination unreachable) and non-error messages (e.g., type = echo). This section applies exclusively to error messages. Disposition of non-error, ICMP messages (that are not addressed to the IPsec implementation itself) MUST be explicitly accounted for using SPD entries.

The discussion in this section applies to ICMPv6 as well as to ICMPv4. Also, a mechanism SHOULD be provided to allow an administrator to cause ICMP error messages (selected, all, or none) to be logged as an aid to problem diagnosis.

6.1. Processing ICMP Error Messages Directed to an IPsec Implementation

6.1.1. ICMP Error Messages Received on the Unprotected Side of the Boundary

Figure 3 in Section 5.2 shows a distinct ICMP processing module on the unprotected side of the IPsec boundary, for processing ICMP messages (error or otherwise) that are addressed to the IPsec device and that are not protected via AH or ESP. An ICMP message of this sort is unauthenticated, and its processing may result in denial or degradation of service. This suggests that, in general, it would be desirable to ignore such messages. However, many ICMP messages will be received by hosts or security gateways from unauthenticated sources, e.g., routers in the public Internet. Ignoring these ICMP messages can degrade service, e.g., because of a failure to process PMTU message and redirection messages. Thus, there is also a motivation for accepting and acting upon unauthenticated ICMP messages.

To accommodate both ends of this spectrum, a compliant IPsec implementation MUST permit a local administrator to configure an IPsec implementation to accept or reject unauthenticated ICMP traffic. This control MUST be at the granularity of ICMP type and MAY be at the granularity of ICMP type and code. Additionally, an implementation SHOULD incorporate mechanisms and parameters for dealing with such traffic. For example, there could be the ability to establish a minimum PMTU for traffic (on a per destination basis), to prevent receipt of an unauthenticated ICMP from setting the PMTU to a trivial size.

If an ICMP PMTU message passes the checks above and the system is configured to accept it, then there are two possibilities. If the implementation applies fragmentation on the ciphertext side of the boundary, then the accepted PMTU information is passed to the forwarding module (outside of the IPsec implementation), which uses it to manage outbound packet fragmentation. If the implementation is configured to effect plaintext side fragmentation, then the PMTU information is passed to the plaintext side and processed as described in Section 8.2.

6.1.2. ICMP Error Messages Received on the Protected Side of the Boundary

These ICMP messages are not authenticated, but they do come from sources on the protected side of the IPsec boundary. Thus, these messages generally are viewed as more "trustworthy" than their counterparts arriving from sources on the unprotected side of the boundary. The major security concern here is that a compromised host or router might emit erroneous ICMP error messages that could degrade service for other devices "behind" the security gateway, or that could even result in violations of confidentiality. For example, if a bogus ICMP redirect were consumed by a security gateway, it could cause the forwarding table on the protected side of the boundary to be modified so as to deliver traffic to an inappropriate destination "behind" the gateway. Thus, implementers MUST provide controls to allow local administrators to constrain the processing of ICMP error messages received on the protected side of the boundary, and directed to the IPsec implementation. These controls are of the same type as those employed on the unprotected side, described above in Section 6.1.1.

6.2. Processing Protected, Transit ICMP Error Messages

When an ICMP error message is transmitted via an SA to a device "behind" an IPsec implementation, both the payload and the header of the ICMP message require checking from an access control perspective. If one of these messages is forwarded to a host behind a security

gateway, the receiving host IP implementation will make decisions based on the payload, i.e., the header of the packet that purportedly triggered the error response. Thus, an IPsec implementation MUST be configurable to check that this payload header information is consistent with the SA via which it arrives. (This means that the payload header, with source and destination address and port fields reversed, matches the traffic selectors for the SA.) If this sort of check is not performed, then, for example, anyone with whom the receiving IPsec system (A) has an active SA could send an ICMP Destination Unreachable message that refers to any host/net with which A is currently communicating, and thus effect a highly efficient DoS attack regarding communication with other peers of A. Normal IPsec receiver processing of traffic is not sufficient to protect against such attacks. However, not all contexts may require such checks, so it is also necessary to allow a local administrator to configure an implementation to NOT perform such checks.

To accommodate both policies, the following convention is adopted. If an administrator wants to allow ICMP error messages to be carried by an SA without inspection of the payload, then configure an SPD entry that explicitly allows for carriage of such traffic. If an administrator wants IPsec to check the payload of ICMP error messages for consistency, then do not create any SPD entries that accommodate carriage of such traffic based on the ICMP packet header. This convention motivates the following processing description.

IPsec senders and receivers MUST support the following processing for ICMP error messages that are sent and received via SAs.

If an SA exists that accommodates an outbound ICMP error message, then the message is mapped to the SA and only the IP and ICMP headers are checked upon receipt, just as would be the case for other traffic. If no SA exists that matches the traffic selectors associated with an ICMP error message, then the SPD is searched to determine if such an SA can be created. If so, the SA is created and the ICMP error message is transmitted via that SA. Upon receipt, this message is subject to the usual traffic selector checks at the receiver. This processing is exactly what would happen for traffic in general, and thus does not represent any special processing for ICMP error messages.

If no SA exists that would carry the outbound ICMP message in question, and if no SPD entry would allow carriage of this outbound ICMP error message, then an IPsec implementation MUST map the message to the SA that would carry the return traffic associated with the packet that triggered the ICMP error message. This requires an IPsec implementation to detect outbound ICMP error messages that map to no extant SA or SPD entry, and treat them specially with regard to SA

creation and lookup. The implementation extracts the header for the packet that triggered the error (from the ICMP message payload), reverses the source and destination IP address fields, extracts the protocol field, and reverses the port fields (if accessible). It then uses this extracted information to locate an appropriate, active outbound SA, and transmits the error message via this SA. If no such SA exists, no SA will be created, and this is an auditable event.

If an IPsec implementation receives an inbound ICMP error message on an SA, and the IP and ICMP headers of the message do not match the traffic selectors for the SA, the receiver MUST process the received message in a special fashion. Specifically, the receiver must extract the header of the triggering packet from the ICMP payload, and reverse fields as described above to determine if the packet is consistent with the selectors for the SA via which the ICMP error message was received. If the packet fails this check, the IPsec implementation MUST NOT forwarded the ICMP message to the destination. This is an auditable event.

7. Handling Fragments (on the protected side of the IPsec boundary)

Earlier sections of this document describe mechanisms for (a) fragmenting an outbound packet after IPsec processing has been applied and reassembling it at the receiver before IPsec processing and (b) handling inbound fragments received from the unprotected side of the IPsec boundary. This section describes how an implementation should handle the processing of outbound plaintext fragments on the protected side of the IPsec boundary. (See Appendix D, "Fragment Handling Rationale".) In particular, it addresses:

- mapping an outbound non-initial fragment to the right SA (or finding the right SPD entry)
- verifying that a received non-initial fragment is authorized for the SA via which it was received
- mapping outbound and inbound non-initial fragments to the right SPD-O/SPD-I entry or the relevant cache entry, for BYPASS/DISCARD traffic

Note: In Section 4.1, transport mode SAs have been defined to not carry fragments (IPv4 or IPv6). Note also that in Section 4.4.1, two special values, ANY and OPAQUE, were defined for selectors and that ANY includes OPAQUE. The term "non-trivial" is used to mean that the selector has a value other than OPAQUE or ANY.

Note: The term "non-initial fragment" is used here to indicate a fragment that does not contain all the selector values that may be needed for access control. As observed in Section 4.4.1, depending on the Next Layer Protocol, in addition to Ports, the ICMP message

type/code or Mobility Header type could be missing from non-initial fragments. Also, for IPv6, even the first fragment might NOT contain the Next Layer Protocol or Ports (or ICMP message type/code, or Mobility Header type) depending on the kind and number of extension headers present. If a non-initial fragment contains the Port (or ICMP type and code or Mobility Header type) but not the Next Layer Protocol, then unless there is an SPD entry for the relevant Local/Remote addresses with ANY for Next Layer Protocol and Port (or ICMP type and code or Mobility Header type), the fragment would not contain all the selector information needed for access control.

To address the above issues, three approaches have been defined:

- Tunnel mode SAs that carry initial and non-initial fragments (See Section 7.1.)
- Separate tunnel mode SAs for non-initial fragments (See Section 7.2.)
- Stateful fragment checking (See Section 7.3.)

7.1. Tunnel Mode SAs that Carry Initial and Non-Initial Fragments

All implementations MUST support tunnel mode SAs that are configured to pass traffic without regard to port field (or ICMP type/code or Mobility Header type) values. If the SA will carry traffic for specified protocols, the selector set for the SA MUST specify the port fields (or ICMP type/code or Mobility Header type) as ANY. An SA defined in this fashion will carry all traffic including initial and non-initial fragments for the indicated Local/Remote addresses and specified Next Layer protocol(s). If the SA will carry traffic without regard to a specific protocol value (i.e., ANY is specified as the (Next Layer) protocol selector value), then the port field values are undefined and MUST be set to ANY as well. (As noted in 4.4.1, ANY includes OPAQUE as well as all specific values.)

7.2. Separate Tunnel Mode SAs for Non-Initial Fragments

An implementation MAY support tunnel mode SAs that will carry only non-initial fragments, separate from non-fragmented packets and initial fragments. The OPAQUE value will be used to specify port (or ICMP type/code or Mobility Header type) field selectors for an SA to carry such fragments. Receivers MUST perform a minimum offset check on IPv4 (non-initial) fragments to protect against overlapping fragment attacks when SAs of this type are employed. Because such checks cannot be performed on IPv6 non-initial fragments, users and administrators are advised that carriage of such fragments may be dangerous, and implementers may choose to NOT support such SAs for IPv6 traffic. Also, an SA of this sort will carry all non-initial fragments that match a specified Local/Remote address pair and

protocol value, i.e., the fragments carried on this SA belong to packets that if not fragmented, might have gone on separate SAs of differing security. Therefore, users and administrators are advised to protect such traffic using ESP (with integrity) and the "strongest" integrity and encryption algorithms in use between both peers. (Determination of the "strongest" algorithms requires imposing an ordering of the available algorithms, a local determination at the discretion of the initiator of the SA.)

Specific port (or ICMP type/code or Mobility Header type) selector values will be used to define SAs to carry initial fragments and non-fragmented packets. This approach can be used if a user or administrator wants to create one or more tunnel mode SAs between the same Local/Remote addresses that discriminate based on port (or ICMP type/code or Mobility Header type) fields. These SAs MUST have non-trivial protocol selector values, otherwise approach #1 above MUST be used.

Note: In general, for the approach described in this section, one needs only a single SA between two implementations to carry all non-initial fragments. However, if one chooses to have multiple SAs between the two implementations for QoS differentiation, then one might also want multiple SAs to carry fragments-without-ports, one for each supported QoS class. Since support for QoS via distinct SAs is a local matter, not mandated by this document, the choice to have multiple SAs to carry non-initial fragments should also be local.

7.3. Stateful Fragment Checking

An implementation MAY support some form of stateful fragment checking for a tunnel mode SA with non-trivial port (or ICMP type/code or MH type) field values (not ANY or OPAQUE). Implementations that will transmit non-initial fragments on a tunnel mode SA that makes use of non-trivial port (or ICMP type/code or MH type) selectors MUST notify a peer via the IKE NOTIFY NON_FIRST_FRAGMENTS_ALSO payload.

The peer MUST reject this proposal if it will not accept non-initial fragments in this context. If an implementation does not successfully negotiate transmission of non-initial fragments for such an SA, it MUST NOT send such fragments over the SA. This standard does not specify how peers will deal with such fragments, e.g., via reassembly or other means, at either sender or receiver. However, a receiver MUST discard non-initial fragments that arrive on an SA with non-trivial port (or ICMP type/code or MH type) selector values unless this feature has been negotiated. Also, the receiver MUST discard non-initial fragments that do not comply with the security policy applied to the overall packet. Discarding such packets is an auditable event. Note that in network configurations where fragments

of a packet might be sent or received via different security gateways or BITW implementations, stateful strategies for tracking fragments may fail.

7.4. BYPASS/DISCARD Traffic

All implementations MUST support DISCARDing of fragments using the normal SPD packet classification mechanisms. All implementations MUST support stateful fragment checking to accommodate BYPASS traffic for which a non-trivial port range is specified. The concern is that BYPASS of a cleartext, non-initial fragment arriving at an IPsec implementation could undermine the security afforded IPsec-protected traffic directed to the same destination. For example, consider an IPsec implementation configured with an SPD entry that calls for IPsec protection of traffic between a specific source/destination address pair, and for a specific protocol and destination port, e.g., TCP traffic on port 23 (Telnet). Assume that the implementation also allows BYPASS of traffic from the same source/destination address pair and protocol, but for a different destination port, e.g., port 119 (NNTP). An attacker could send a non-initial fragment (with a forged source address) that, if bypassed, could overlap with IPsec-protected traffic from the same source and thus violate the integrity of the IPsec-protected traffic. Requiring stateful fragment checking for BYPASS entries with non-trivial port ranges prevents attacks of this sort. As noted above, in network configurations where fragments of a packet might be sent or received via different security gateways or BITW implementations, stateful strategies for tracking fragments may fail.

8. Path MTU/DF Processing

The application of AH or ESP to an outbound packet increases the size of a packet and thus may cause a packet to exceed the PMTU for the SA via which the packet will travel. An IPsec implementation also may receive an unprotected ICMP PMTU message and, if it chooses to act upon the message, the result will affect outbound traffic processing. This section describes the processing required of an IPsec implementation to deal with these two PMTU issues.

8.1. DF Bit

All IPsec implementations MUST support the option of copying the DF bit from an outbound packet to the tunnel mode header that it emits, when traffic is carried via a tunnel mode SA. This means that it MUST be possible to configure the implementation’s treatment of the DF bit (set, clear, copy from inner header) for each SA. This applies to SAs where both inner and outer headers are IPv4.

8.2. Path MTU (PMTU) Discovery

This section discusses IPsec handling for unprotected Path MTU Discovery messages. ICMP PMTU is used here to refer to an ICMP message for:

IPv4 (RFC 792 [Pos81b]):
- Type = 3 (Destination Unreachable)
- Code = 4 (Fragmentation needed and DF set)
- Next-Hop MTU in the low-order 16 bits of the second word of the ICMP header (labeled "unused" in RFC 792), with high-order 16 bits set to zero)

IPv6 (RFC 2463 [CD98]):
- Type = 2 (Packet Too Big)
- Code = 0 (Fragmentation needed)
- Next-Hop MTU in the 32-bit MTU field of the ICMP6 message

8.2.1. Propagation of PMTU

When an IPsec implementation receives an unauthenticated PMTU message, and it is configured to process (vs. ignore) such messages, it maps the message to the SA to which it corresponds. This mapping is effected by extracting the header information from the payload of the PMTU message and applying the procedure described in Section 5.2. The PMTU determined by this message is used to update the SAD PMTU field, taking into account the size of the AH or ESP header that will be applied, any crypto synchronization data, and the overhead imposed by an additional IP header, in the case of a tunnel mode SA.

In a native host implementation, it is possible to maintain PMTU data at the same granularity as for unprotected communication, so there is no loss of functionality. Signaling of the PMTU information is internal to the host. For all other IPsec implementation options, the PMTU data must be propagated via a synthesized ICMP PMTU. In these cases, the IPsec implementation SHOULD wait for outbound traffic to be mapped to the SAD entry. When such traffic arrives, if the traffic would exceed the updated PMTU value the traffic MUST be handled as follows:

Case 1: Original (cleartext) packet is IPv4 and has the DF bit set. The implementation SHOULD discard the packet and send a PMTU ICMP message.

Case 2: Original (cleartext) packet is IPv4 and has the DF bit clear. The implementation SHOULD fragment (before or after encryption per its configuration) and then forward the fragments. It SHOULD NOT send a PMTU ICMP message.

Case 3: Original (cleartext) packet is IPv6. The implementation SHOULD discard the packet and send a PMTU ICMP message.

8.2.2. PMTU Aging

In all IPsec implementations, the PMTU associated with an SA MUST be "aged" and some mechanism is required to update the PMTU in a timely manner, especially for discovering if the PMTU is smaller than required by current network conditions. A given PMTU has to remain in place long enough for a packet to get from the source of the SA to the peer, and to propagate an ICMP error message if the current PMTU is too big.

Implementations SHOULD use the approach described in the Path MTU Discovery document (RFC 1191 [MD90], Section 6.3), which suggests periodically resetting the PMTU to the first-hop data-link MTU and then letting the normal PMTU Discovery processes update the PMTU as necessary. The period SHOULD be configurable.

9. Auditing

IPsec implementations are not required to support auditing. For the most part, the granularity of auditing is a local matter. However, several auditable events are identified in this document, and for each of these events a minimum set of information that SHOULD be included in an audit log is defined. Additional information also MAY be included in the audit log for each of these events, and additional events, not explicitly called out in this specification, also MAY result in audit log entries. There is no requirement for the receiver to transmit any message to the purported transmitter in response to the detection of an auditable event, because of the potential to induce denial of service via such action.

10. Conformance Requirements

All IPv4 IPsec implementations MUST comply with all requirements of this document. All IPv6 implementations MUST comply with all requirements of this document.

11. Security Considerations

The focus of this document is security; hence security considerations permeate this specification.

IPsec imposes stringent constraints on bypass of IP header data in both directions, across the IPsec barrier, especially when tunnel mode SAs are employed. Some constraints are absolute, while others are subject to local administrative controls, often on a per-SA basis. For outbound traffic, these constraints are designed to limit covert channel bandwidth. For inbound traffic, the constraints are designed to prevent an adversary who has the ability to tamper with one data stream (on the unprotected side of the IPsec barrier) from adversely affecting other data streams (on the protected side of the barrier). The discussion in Section 5 dealing with processing DSCP values for tunnel mode SAs illustrates this concern.

If an IPsec implementation is configured to pass ICMP error messages over SAs based on the ICMP header values, without checking the header information from the ICMP message payload, serious vulnerabilities may arise. Consider a scenario in which several sites (A, B, and C) are connected to one another via ESP-protected tunnels: A-B, A-C, and B-C. Also assume that the traffic selectors for each tunnel specify ANY for protocol and port fields and IP source/destination address ranges that encompass the address range for the systems behind the security gateways serving each site. This would allow a host at site B to send an ICMP Destination Unreachable message to any host at site A, that declares all hosts on the net at site C to be unreachable. This is a very efficient DoS attack that could have been prevented if the ICMP error messages were subjected to the checks that IPsec provides, if the SPD is suitably configured, as described in Section 6.2.

12. IANA Considerations

The IANA has assigned the value (3) for the asn1-modules registry and has assigned the object identifier 1.3.6.1.5.8.3.1 for the SPD module. See Appendix C, "ASN.1 for an SPD Entry".

13. Differences from RFC 2401

This architecture document differs substantially from RFC 2401 [RFC2401] in detail and in organization, but the fundamental notions are unchanged.

- The processing model has been revised to address new IPsec scenarios, improve performance, and simplify implementation. This includes a separation between forwarding (routing) and SPD

selection, several SPD changes, and the addition of an outbound SPD cache and an inbound SPD cache for bypassed or discarded traffic. There is also a new database, the Peer Authorization Database (PAD). This provides a link between an SA management protocol (such as IKE) and the SPD.

- o There is no longer a requirement to support nested SAs or "SA bundles". Instead this functionality can be achieved through SPD and forwarding table configuration. An example of a configuration has been added in Appendix E.

- o SPD entries were redefined to provide more flexibility. Each SPD entry now consists of 1 to N sets of selectors, where each selector set contains one protocol and a "list of ranges" can now be specified for the Local IP address, Remote IP address, and whatever fields (if any) are associated with the Next Layer Protocol (Local Port, Remote Port, ICMP message type and code, and Mobility Header type). An individual value for a selector is represented via a trivial range and ANY is represented via a range than spans all values for the selector. An example of an ASN.1 description is included in Appendix C.

- o TOS (IPv4) and Traffic Class (IPv6) have been replaced by DSCP and ECN. The tunnel section has been updated to explain how to handle DSCP and ECN bits.

- o For tunnel mode SAs, an SG, BITS, or BITW implementation is now allowed to fragment packets before applying IPsec. This applies only to IPv4. For IPv6 packets, only the originator is allowed to fragment them.

- o When security is desired between two intermediate systems along a path or between an intermediate system and an end system, transport mode may now be used between security gateways and between a security gateway and a host.

- o This document clarifies that for all traffic that crosses the IPsec boundary, including IPsec management traffic, the SPD or associated caches must be consulted.

- o This document defines how to handle the situation of a security gateway with multiple subscribers requiring separate IPsec contexts.

- o A definition of reserved SPIs has been added.

- Text has been added explaining why ALL IP packets must be checked -- IPsec includes minimal firewall functionality to support access control at the IP layer.

- The tunnel section has been updated to clarify how to handle the IP options field and IPv6 extension headers when constructing the outer header.

- SA mapping for inbound traffic has been updated to be consistent with the changes made in AH and ESP for support of unicast and multicast SAs.

- Guidance has been added regarding how to handle the covert channel created in tunnel mode by copying the DSCP value to outer header.

- Support for AH in both IPv4 and IPv6 is no longer required.

- PMTU handling has been updated. The appendix on PMTU/DF/Fragmentation has been deleted.

- Three approaches have been added for handling plaintext fragments on the protected side of the IPsec boundary. Appendix D documents the rationale behind them.

- Added revised text describing how to derive selector values for SAs (from the SPD entry or from the packet, etc.)

- Added a new table describing the relationship between selector values in an SPD entry, the PFP flag, and resulting selector values in the corresponding SAD entry.

- Added Appendix B to describe decorrelation.

- Added text describing how to handle an outbound packet that must be discarded.

- Added text describing how to handle a DISCARDED inbound packet, i.e., one that does not match the SA upon which it arrived.

- IPv6 mobility header has been added as a possible Next Layer Protocol. IPv6 Mobility Header message type has been added as a selector.

- ICMP message type and code have been added as selectors.

- The selector "data sensitivity level" has been removed to simplify things.

- Updated text describing handling ICMP error messages. The appendix on "Categorization of ICMP Messages" has been deleted.
- The text for the selector name has been updated and clarified.
- The "Next Layer Protocol" has been further explained and a default list of protocols to skip when looking for the Next Layer Protocol has been added.
- The text has been amended to say that this document assumes use of IKEv2 or an SA management protocol with comparable features.
- Text has been added clarifying the algorithm for mapping inbound IPsec datagrams to SAs in the presence of multicast SAs.
- The appendix "Sequence Space Window Code Example" has been removed.
- With respect to IP addresses and ports, the terms "Local" and "Remote" are used for policy rules (replacing source and destination). "Local" refers to the entity being protected by an IPsec implementation, i.e., the "source" address/port of outbound packets or the "destination" address/port of inbound packets. "Remote" refers to a peer entity or peer entities. The terms "source" and "destination" are still used for packet header fields.

14. Acknowledgements

The authors would like to acknowledge the contributions of Ran Atkinson, who played a critical role in initial IPsec activities, and who authored the first series of IPsec standards: RFCs 1825-1827; and Charlie Lynn, who made significant contributions to the second series of IPsec standards (RFCs 2401, 2402, and 2406) and to the current versions, especially with regard to IPv6 issues. The authors also would like to thank the members of the IPsec and MSEC working groups who have contributed to the development of this protocol specification.

Appendix A: Glossary

This section provides definitions for several key terms that are employed in this document. Other documents provide additional definitions and background information relevant to this technology, e.g., [Shi00], [VK83], and [HA94]. Included in this glossary are generic security service and security mechanism terms, plus IPsec-specific terms.

Access Control
: A security service that prevents unauthorized use of a resource, including the prevention of use of a resource in an unauthorized manner. In the IPsec context, the resource to which access is being controlled is often:

  - for a host, computing cycles or data
  - for a security gateway, a network behind the gateway or bandwidth on that network.

Anti-replay
: See "Integrity" below.

Authentication
: Used informally to refer to the combination of two nominally distinct security services, data origin authentication and connectionless integrity. See the definitions below for each of these services.

Availability
: When viewed as a security service, addresses the security concerns engendered by attacks against networks that deny or degrade service. For example, in the IPsec context, the use of anti-replay mechanisms in AH and ESP support availability.

Confidentiality
: The security service that protects data from unauthorized disclosure. The primary confidentiality concern in most instances is unauthorized disclosure of application-level data, but disclosure of the external characteristics of communication also can be a concern in some circumstances. Traffic flow confidentiality is the service that addresses this latter concern by concealing source and destination addresses, message length, or frequency of communication. In the IPsec context, using ESP in tunnel mode, especially at a security gateway, can provide some level of traffic flow confidentiality. (See also "Traffic Analysis" below.)

Data Origin Authentication
A security service that verifies the identity of the claimed source of data. This service is usually bundled with connectionless integrity service.

Encryption
A security mechanism used to transform data from an intelligible form (plaintext) into an unintelligible form (ciphertext), to provide confidentiality. The inverse transformation process is designated "decryption". Often the term "encryption" is used to generically refer to both processes.

Integrity
A security service that ensures that modifications to data are detectable. Integrity comes in various flavors to match application requirements. IPsec supports two forms of integrity: connectionless and a form of partial sequence integrity. Connectionless integrity is a service that detects modification of an individual IP datagram, without regard to the ordering of the datagram in a stream of traffic. The form of partial sequence integrity offered in IPsec is referred to as anti-replay integrity, and it detects arrival of duplicate IP datagrams (within a constrained window). This is in contrast to connection-oriented integrity, which imposes more stringent sequencing requirements on traffic, e.g., to be able to detect lost or re-ordered messages. Although authentication and integrity services often are cited separately, in practice they are intimately connected and almost always offered in tandem.

Protected vs. Unprotected
"Protected" refers to the systems or interfaces that are inside the IPsec protection boundary, and "unprotected" refers to the systems or interfaces that are outside the IPsec protection boundary. IPsec provides a boundary through which traffic passes. There is an asymmetry to this barrier, which is reflected in the processing model. Outbound data, if not discarded or bypassed, is protected via the application of AH or ESP and the addition of the corresponding headers. Inbound data, if not discarded or bypassed, is processed via the removal of AH or ESP headers. In this document, inbound traffic enters an IPsec implementation from the "unprotected" interface. Outbound traffic enters the implementation via the "protected" interface, or is internally generated by the implementation on the "protected" side of the boundary and directed toward the "unprotected" interface. An IPsec implementation may support more than one interface on either or both sides of the boundary. The protected interface may be

internal, e.g., in a host implementation of IPsec. The protected interface may link to a socket layer interface presented by the OS.

Security Association (SA)
A simplex (uni-directional) logical connection, created for security purposes. All traffic traversing an SA is provided the same security processing. In IPsec, an SA is an Internet-layer abstraction implemented through the use of AH or ESP. State data associated with an SA is represented in the SA Database (SAD).

Security Gateway
An intermediate system that acts as the communications interface between two networks. The set of hosts (and networks) on the external side of the security gateway is termed unprotected (they are generally at least less protected than those "behind" the SG), while the networks and hosts on the internal side are viewed as protected. The internal subnets and hosts served by a security gateway are presumed to be trusted by virtue of sharing a common, local, security administration. In the IPsec context, a security gateway is a point at which AH and/or ESP is implemented in order to serve a set of internal hosts, providing security services for these hosts when they communicate with external hosts also employing IPsec (either directly or via another security gateway).

Security Parameters Index (SPI)
An arbitrary 32-bit value that is used by a receiver to identify the SA to which an incoming packet should be bound. For a unicast SA, the SPI can be used by itself to specify an SA, or it may be used in conjunction with the IPsec protocol type. Additional IP address information is used to identify multicast SAs. The SPI is carried in AH and ESP protocols to enable the receiving system to select the SA under which a received packet will be processed. An SPI has only local significance, as defined by the creator of the SA (usually the receiver of the packet carrying the SPI); thus an SPI is generally viewed as an opaque bit string. However, the creator of an SA may choose to interpret the bits in an SPI to facilitate local processing.

Traffic Analysis
The analysis of network traffic flow for the purpose of deducing information that is useful to an adversary. Examples of such information are frequency of transmission, the identities of the conversing parties, sizes of packets, and flow identifiers [Sch94].

Appendix B: Decorrelation

This appendix is based on work done for caching of policies in the IP Security Policy Working Group by Luis Sanchez, Matt Condell, and John Zao.

Two SPD entries are correlated if there is a non-null intersection between the values of corresponding selectors in each entry. Caching correlated SPD entries can lead to incorrect policy enforcement. A solution to this problem, which still allows for caching, is to remove the ambiguities by decorrelating the entries. That is, the SPD entries must be rewritten so that for every pair of entries there exists a selector for which there is a null intersection between the values in both of the entries. Once the entries are decorrelated, there is no longer any ordering requirement on them, since only one entry will match any lookup. The next section describes decorrelation in more detail and presents an algorithm that may be used to implement decorrelation.

B.1. Decorrelation Algorithm

The basic decorrelation algorithm takes each entry in a correlated SPD and divides it into a set of entries using a tree structure. The nodes of the tree are the selectors that may overlap between the policies. At each node, the algorithm creates a branch for each of the values of the selector. It also creates one branch for the complement of the union of all selector values. Policies are then formed by traversing the tree from the root to each leaf. The policies at the leaves are compared to the set of already decorrelated policy rules. Each policy at a leaf is either completely overridden by a policy in the already decorrelated set and is discarded or is decorrelated with all the policies in the decorrelated set and is added to it.

The basic algorithm does not guarantee an optimal set of decorrelated entries. That is, the entries may be broken up into smaller sets than is necessary, though they will still provide all the necessary policy information. Some extensions to the basic algorithm are described later to improve this and improve the performance of the algorithm.

C A set of ordered, correlated entries (a correlated SPD).
Ci The ith entry in C.
U The set of decorrelated entries being built from C.
Ui The ith entry in U.
Sik The kth selection for policy Ci.
Ai The action for policy Ci.

A policy (SPD entry) P may be expressed as a sequence of selector values and an action (BYPASS, DISCARD, or PROTECT):

Ci = Si1 x Si2 x ... x Sik -> Ai

1) Put C1 in set U as U1

For each policy Cj (j > 1) in C

2) If Cj is decorrelated with every entry in U, then add it to U.

3) If Cj is correlated with one or more entries in U, create a tree rooted at the policy Cj that partitions Cj into a set of decorrelated entries. The algorithm starts with a root node where no selectors have yet been chosen.

A) Choose a selector in Cj, Sjn, that has not yet been chosen when traversing the tree from the root to this node. If there are no selectors not yet used, continue to the next unfinished branch until all branches have been completed. When the tree is completed, go to step D.

   T is the set of entries in U that are correlated with the entry at this node.

   The entry at this node is the entry formed by the selector values of each of the branches between the root and this node. Any selector values that are not yet represented by branches assume the corresponding selector value in Cj, since the values in Cj represent the maximum value for each selector.

B) Add a branch to the tree for each value of the selector Sjn that appears in any of the entries in T. (If the value is a superset of the value of Sjn in Cj, then use the value in Cj, since that value represents the universal set.) Also add a branch for the complement of the union of all the values of the selector Sjn in T. When taking the complement, remember that the universal set is the value of Sjn in Cj. A branch need not be created for the null set.

C) Repeat A and B until the tree is completed.

D) The entry to each leaf now represents an entry that is a subset of Cj. The entries at the leaves completely partition Cj in such a way that each entry is either completely overridden by an entry in U, or is decorrelated with the entries in U.

   Add all the decorrelated entries at the leaves of the tree to U.

4) Get next Cj and go to 2.

5) When all entries in C have been processed, then U will contain an decorrelated version of C.

There are several optimizations that can be made to this algorithm. A few of them are presented here.

It is possible to optimize, or at least improve, the amount of branching that occurs by carefully choosing the order of the selectors used for the next branch. For example, if a selector Sjn can be chosen so that all the values for that selector in T are equal to or a superset of the value of Sjn in Cj, then only a single branch needs to be created (since the complement will be null).

Branches of the tree do not have to proceed with the entire decorrelation algorithm. For example, if a node represents an entry that is decorrelated with all the entries in U, then there is no reason to continue decorrelating that branch. Also, if a branch is completely overridden by an entry in U, then there is no reason to continue decorrelating the branch.

An additional optimization is to check to see if a branch is overridden by one of the CORRELATED entries in set C that has already been decorrelated. That is, if the branch is part of decorrelating Cj, then check to see if it was overridden by an entry Cm, m < j. This is a valid check, since all the entries Cm are already expressed in U.

Along with checking if an entry is already decorrelated in step 2, check if Cj is overridden by any entry in U. If it is, skip it since it is not relevant. An entry x is overridden by another entry y if every selector in x is equal to or a subset of the corresponding selector in entry y.

Appendix C: ASN.1 for an SPD Entry

This appendix is included as an additional way to describe SPD entries, as defined in Section 4.4.1. It uses ASN.1 syntax that has been successfully compiled. This syntax is merely illustrative and need not be employed in an implementation to achieve compliance. The SPD description in Section 4.4.1 is normative.

```
SPDModule

 {iso(1) org (3) dod (6) internet (1) security (5) mechanisms (5)
  ipsec (8) asn1-modules (3) spd-module (1) }

    DEFINITIONS IMPLICIT TAGS ::=

    BEGIN

    IMPORTS
        RDNSequence FROM PKIX1Explicit88
          { iso(1) identified-organization(3)
            dod(6) internet(1) security(5) mechanisms(5) pkix(7)
            id-mod(0) id-pkix1-explicit(18) } ;

    -- An SPD is a list of policies in decreasing order of preference
    SPD ::= SEQUENCE OF SPDEntry

    SPDEntry ::= CHOICE {
        iPsecEntry       IPsecEntry,               -- PROTECT traffic
        bypassOrDiscard  [0] BypassOrDiscardEntry } -- DISCARD/BYPASS

    IPsecEntry ::= SEQUENCE {       -- Each entry consists of
        name         NameSets OPTIONAL,
        pFPs         PacketFlags,    -- Populate from packet flags
                            -- Applies to ALL of the corresponding
                            -- traffic selectors in the SelectorLists
        condition    SelectorLists,  -- Policy "condition"
        processing   Processing      -- Policy "action"
        }

    BypassOrDiscardEntry ::= SEQUENCE {
        bypass       BOOLEAN,        -- TRUE BYPASS, FALSE DISCARD
        condition    InOutBound }

    InOutBound ::= CHOICE {
        outbound     [0] SelectorLists,
        inbound      [1] SelectorLists,
        bothways     [2] BothWays }
```

```
BothWays ::= SEQUENCE {
    inbound     SelectorLists,
    outbound    SelectorLists }

NameSets ::= SEQUENCE {
    passed      SET OF Names-R,  -- Matched to IKE ID by
                                 -- responder
    local       SET OF Names-I } -- Used internally by IKE
                                 -- initiator

Names-R ::= CHOICE {                    -- IKEv2 IDs
    dName       RDNSequence,            -- ID_DER_ASN1_DN
    fqdn        FQDN,                   -- ID_FQDN
    rfc822      [0] RFC822Name,         -- ID_RFC822_ADDR
    keyID       OCTET STRING }          -- KEY_ID

Names-I ::= OCTET STRING       -- Used internally by IKE
                               -- initiator

FQDN ::= IA5String

RFC822Name ::= IA5String

PacketFlags ::= BIT STRING {
            -- if set, take selector value from packet
            -- establishing SA
            -- else use value in SPD entry
    localAddr  (0),
    remoteAddr (1),
    protocol   (2),
    localPort  (3),
    remotePort (4)  }

SelectorLists ::= SET OF SelectorList

SelectorList ::= SEQUENCE {
    localAddr   AddrList,
    remoteAddr  AddrList,
    protocol    ProtocolChoice }

Processing ::= SEQUENCE {
    extSeqNum   BOOLEAN, -- TRUE 64 bit counter, FALSE 32 bit
    seqOverflow BOOLEAN, -- TRUE rekey, FALSE terminate & audit
    fragCheck   BOOLEAN, -- TRUE stateful fragment checking,
                         -- FALSE no stateful fragment checking
    lifetime    SALifetime,
    spi         ManualSPI,
    algorithms  ProcessingAlgs,
```

```
        tunnel      TunnelOptions OPTIONAL } -- if absent, use
                                             -- transport mode

    SALifetime ::= SEQUENCE {
        seconds   [0] INTEGER OPTIONAL,
        bytes     [1] INTEGER OPTIONAL }

    ManualSPI ::= SEQUENCE {
        spi     INTEGER,
        keys    KeyIDs }

    KeyIDs ::= SEQUENCE OF OCTET STRING

    ProcessingAlgs ::= CHOICE {
        ah          [0] IntegrityAlgs,  -- AH
        esp         [1] ESPAlgs}        -- ESP

    ESPAlgs ::= CHOICE {
        integrity       [0] IntegrityAlgs,       -- integrity only
        confidentiality [1] ConfidentialityAlgs, -- confidentiality
                                                 -- only
        both            [2] IntegrityConfidentialityAlgs,
        combined        [3] CombinedModeAlgs }

    IntegrityConfidentialityAlgs ::= SEQUENCE {
        integrity       IntegrityAlgs,
        confidentiality ConfidentialityAlgs }

    -- Integrity Algorithms, ordered by decreasing preference
    IntegrityAlgs ::= SEQUENCE OF IntegrityAlg

    -- Confidentiality Algorithms, ordered by decreasing preference
    ConfidentialityAlgs ::= SEQUENCE OF ConfidentialityAlg

    -- Integrity Algorithms
    IntegrityAlg ::= SEQUENCE {
        algorithm   IntegrityAlgType,
        parameters  ANY -- DEFINED BY algorithm -- OPTIONAL }

    IntegrityAlgType ::= INTEGER {
        none              (0),
        auth-HMAC-MD5-96  (1),
        auth-HMAC-SHA1-96 (2),
        auth-DES-MAC      (3),
        auth-KPDK-MD5     (4),
        auth-AES-XCBC-96  (5)
    --  tbd (6..65535)
        }
```

```
-- Confidentiality Algorithms
ConfidentialityAlg ::= SEQUENCE {
    algorithm   ConfidentialityAlgType,
    parameters  ANY -- DEFINED BY algorithm -- OPTIONAL }

ConfidentialityAlgType ::= INTEGER {
    encr-DES-IV64   (1),
    encr-DES        (2),
    encr-3DES       (3),
    encr-RC5        (4),
    encr-IDEA       (5),
    encr-CAST       (6),
    encr-BLOWFISH   (7),
    encr-3IDEA      (8),
    encr-DES-IV32   (9),
    encr-RC4       (10),
    encr-NULL      (11),
    encr-AES-CBC   (12),
    encr-AES-CTR   (13)
--  tbd (14..65535)
    }

CombinedModeAlgs ::= SEQUENCE OF CombinedModeAlg

CombinedModeAlg ::= SEQUENCE {
    algorithm   CombinedModeType,
    parameters  ANY -- DEFINED BY algorithm} -- defined outside
                                -- of this document for AES modes.

CombinedModeType ::= INTEGER {
    comb-AES-CCM    (1),
    comb-AES-GCM    (2)
--  tbd (3..65535)
    }

TunnelOptions ::= SEQUENCE {
    dscp        DSCP,
    ecn         BOOLEAN,    -- TRUE Copy CE to inner header
    df          DF,
    addresses   TunnelAddresses }

TunnelAddresses ::= CHOICE {
    ipv4        IPv4Pair,
    ipv6        [0] IPv6Pair }

IPv4Pair ::= SEQUENCE {
    local       OCTET STRING (SIZE(4)),
    remote      OCTET STRING (SIZE(4)) }
```

```
IPv6Pair ::= SEQUENCE {
    local       OCTET STRING (SIZE(16)),
    remote      OCTET STRING (SIZE(16)) }

DSCP ::= SEQUENCE {
    copy      BOOLEAN, -- TRUE copy from inner header
                       -- FALSE do not copy
    mapping   OCTET STRING OPTIONAL} -- points to table
                                     -- if no copy

DF ::= INTEGER {
    clear   (0),
    set     (1),
    copy    (2) }

ProtocolChoice::= CHOICE {
    anyProt  AnyProtocol,              -- for ANY protocol
    noNext   [0] NoNextLayerProtocol,  -- has no next layer
                                       -- items
    oneNext  [1] OneNextLayerProtocol, -- has one next layer
                                       -- item
    twoNext  [2] TwoNextLayerProtocol, -- has two next layer
                                       -- items
    fragment FragmentNoNext }          -- has no next layer
                                       -- info

AnyProtocol ::= SEQUENCE {
    id          INTEGER (0),    -- ANY protocol
    nextLayer   AnyNextLayers }

AnyNextLayers ::= SEQUENCE {      -- with either
    first       AnyNextLayer,     -- ANY next layer selector
    second      AnyNextLayer }    -- ANY next layer selector

NoNextLayerProtocol ::= INTEGER (2..254)

FragmentNoNext ::= INTEGER (44)   -- Fragment identifier

OneNextLayerProtocol ::= SEQUENCE {
    id          INTEGER (1..254),   -- ICMP, MH, ICMPv6
    nextLayer   NextLayerChoice }   -- ICMP Type*256+Code
                                    -- MH   Type*256

TwoNextLayerProtocol ::= SEQUENCE {
    id          INTEGER (2..254),   -- Protocol
    local       NextLayerChoice,    -- Local and
    remote      NextLayerChoice }   -- Remote ports
```

```
NextLayerChoice ::= CHOICE {
    any         AnyNextLayer,
    opaque      [0] OpaqueNextLayer,
    range       [1] NextLayerRange }

-- Representation of ANY in next layer field
AnyNextLayer ::= SEQUENCE {
    start       INTEGER (0),
    end         INTEGER (65535) }

-- Representation of OPAQUE in next layer field.
-- Matches IKE convention
OpaqueNextLayer ::= SEQUENCE {
    start       INTEGER (65535),
    end         INTEGER (0) }

-- Range for a next layer field
NextLayerRange ::= SEQUENCE {
    start       INTEGER (0..65535),
    end         INTEGER (0..65535) }

-- List of IP addresses
AddrList ::= SEQUENCE {
    v4List      IPv4List OPTIONAL,
    v6List      [0] IPv6List OPTIONAL }

-- IPv4 address representations
IPv4List ::= SEQUENCE OF IPv4Range

IPv4Range ::= SEQUENCE {    -- close, but not quite right ...
    ipv4Start   OCTET STRING (SIZE (4)),
    ipv4End     OCTET STRING (SIZE (4)) }

-- IPv6 address representations
IPv6List ::= SEQUENCE OF IPv6Range

IPv6Range ::= SEQUENCE {    -- close, but not quite right ...
    ipv6Start   OCTET STRING (SIZE (16)),
    ipv6End     OCTET STRING (SIZE (16)) }

END
```

Appendix D: Fragment Handling Rationale

There are three issues that must be resolved regarding processing of (plaintext) fragments in IPsec:

- mapping a non-initial, outbound fragment to the right SA (or finding the right SPD entry)
- verifying that a received, non-initial fragment is authorized for the SA via which it is received
- mapping outbound and inbound non-initial fragments to the right SPD/cache entry, for BYPASS/DISCARD traffic

The first and third issues arise because we need a deterministic algorithm for mapping traffic to SAs (and SPD/cache entries). All three issues are important because we want to make sure that non-initial fragments that cross the IPsec boundary do not cause the access control policies in place at the receiver (or transmitter) to be violated.

D.1. Transport Mode and Fragments

First, we note that transport mode SAs have been defined to not carry fragments. This is a carryover from RFC 2401, where transport mode SAs always terminated at endpoints. This is a fundamental requirement because, in the worst case, an IPv4 fragment to which IPsec was applied might then be fragmented (as a ciphertext packet), en route to the destination. IP fragment reassembly procedures at the IPsec receiver would not be able to distinguish between pre-IPsec fragments and fragments created after IPsec processing.

For IPv6, only the sender is allowed to fragment a packet. As for IPv4, an IPsec implementation is allowed to fragment tunnel mode packets after IPsec processing, because it is the sender relative to the (outer) tunnel header. However, unlike IPv4, it would be feasible to carry a plaintext fragment on a transport mode SA, because the fragment header in IPv6 would appear after the AH or ESP header, and thus would not cause confusion at the receiver with respect to reassembly. Specifically, the receiver would not attempt reassembly for the fragment until after IPsec processing. To keep things simple, this specification prohibits carriage of fragments on transport mode SAs for IPv6 traffic.

When only end systems used transport mode SAs, the prohibition on carriage of fragments was not a problem, since we assumed that the end system could be configured to not offer a fragment to IPsec. For a native host implementation, this seems reasonable, and, as someone already noted, RFC 2401 warned that a BITS implementation might have to reassemble fragments before performing an SA lookup. (It would

then apply AH or ESP and could re-fragment the packet after IPsec processing.) Because a BITS implementation is assumed to be able to have access to all traffic emanating from its host, even if the host has multiple interfaces, this was deemed a reasonable mandate.

In this specification, it is acceptable to use transport mode in cases where the IPsec implementation is not the ultimate destination, e.g., between two SGs. In principle, this creates a new opportunity for outbound, plaintext fragments to be mapped to a transport mode SA for IPsec processing. However, in these new contexts in which a transport mode SA is now approved for use, it seems likely that we can continue to prohibit transmission of fragments, as seen by IPsec, i.e., packets that have an "outer header" with a non-zero fragment offset field. For example, in an IP overlay network, packets being sent over transport mode SAs are IP-in-IP tunneled and thus have the necessary inner header to accommodate fragmentation prior to IPsec processing. When carried via a transport mode SA, IPsec would not examine the inner IP header for such traffic, and thus would not consider the packet to be a fragment.

D.2. Tunnel Mode and Fragments

For tunnel mode SAs, it has always been the case that outbound fragments might arrive for processing at an IPsec implementation. The need to accommodate fragmented outbound packets can pose a problem because a non-initial fragment generally will not contain the port fields associated with a next layer protocol such as TCP, UDP, or SCTP. Thus, depending on the SPD configuration for a given IPsec implementation, plaintext fragments might or might not pose a problem.

For example, if the SPD requires that all traffic between two address ranges is offered IPsec protection (no BYPASS or DISCARD SPD entries apply to this address range), then it should be easy to carry non-initial fragments on the SA defined for this address range, since the SPD entry implies an intent to carry ALL traffic between the address ranges. But, if there are multiple SPD entries that could match a fragment, and if these entries reference different subsets of port fields (vs. ANY), then it is not possible to map an outbound non-initial fragment to the right entry, unambiguously. (If we choose to allow carriage of fragments on transport mode SAs for IPv6, the problems arises in that context as well.)

This problem largely, though not exclusively, motivated the definition of OPAQUE as a selector value for port fields in RFC 2401. The other motivation for OPAQUE is the observation that port fields might not be accessible due to the prior application of IPsec. For example, if a host applied IPsec to its traffic and that traffic

arrived at an SG, these fields would be encrypted. The algorithm specified for locating the "next layer protocol" described in RFC 2401 also motivated use of OPAQUE to accommodate an encrypted next layer protocol field in such circumstances. Nonetheless, the primary use of the OPAQUE value was to match traffic selector fields in packets that did not contain port fields (non-initial fragments), or packets in which the port fields were already encrypted (as a result of nested application of IPsec). RFC 2401 was ambiguous in discussing the use of OPAQUE vs. ANY, suggesting in some places that ANY might be an alternative to OPAQUE.

We gain additional access control capability by defining both ANY and OPAQUE values. OPAQUE can be defined to match only fields that are not accessible. We could define ANY as the complement of OPAQUE, i.e., it would match all values but only for accessible port fields. We have therefore simplified the procedure employed to locate the next layer protocol in this document, so that we treat ESP and AH as next layer protocols. As a result, the notion of an encrypted next layer protocol field has vanished, and there is also no need to worry about encrypted port fields either. And accordingly, OPAQUE will be applicable only to non-initial fragments.

Since we have adopted the definitions above for ANY and OPAQUE, we need to clarify how these values work when the specified protocol does not have port fields, and when ANY is used for the protocol selector. Accordingly, if a specific protocol value is used as a selector, and if that protocol has no port fields, then the port field selectors are to be ignored and ANY MUST be specified as the value for the port fields. (In this context, ICMP TYPE and CODE values are lumped together as a single port field (for IKEv2 negotiation), as is the IPv6 Mobility Header TYPE value.) If the protocol selector is ANY, then this should be treated as equivalent to specifying a protocol for which no port fields are defined, and thus the port selectors should be ignored, and MUST be set to ANY.

D.3. The Problem of Non-Initial Fragments

For an SG implementation, it is obvious that fragments might arrive from end systems behind the SG. A BITW implementation also may encounter fragments from a host or gateway behind it. (As noted earlier, native host implementations and BITS implementations probably can avoid the problems described below.) In the worst case, fragments from a packet might arrive at distinct BITW or SG instantiations and thus preclude reassembly as a solution option. Hence, in RFC 2401 we adopted a general requirement that fragments must be accommodated in tunnel mode for all implementations. However,

RFC 2401 did not provide a perfect solution. The use of OPAQUE as a selector value for port fields (a SHOULD in RFC 2401) allowed an SA to carry non-initial fragments.

Using the features defined in RFC 2401, if one defined an SA between two IPsec (SG or BITW) implementations using the OPAQUE value for both port fields, then all non-initial fragments matching the source/destination (S/D) address and protocol values for the SA would be mapped to that SA. Initial fragments would NOT map to this SA, if we adopt a strict definition of OPAQUE. However, RFC 2401 did not provide detailed guidance on this and thus it may not have been apparent that use of this feature would essentially create a "non-initial fragment only" SA.

In the course of discussing the "fragment-only" SA approach, it was noted that some subtle problems, problems not considered in RFC 2401, would have to be avoided. For example, an SA of this sort must be configured to offer the "highest quality" security services for any traffic between the indicated S/D addresses (for the specified protocol). This is necessary to ensure that any traffic captured by the fragment-only SA is not offered degraded security relative to what it would have been offered if the packet were not fragmented. A possible problem here is that we may not be able to identify the "highest quality" security services defined for use between two IPsec implementation, since the choice of security protocols, options, and algorithms is a lattice, not a totally ordered set. (We might safely say that BYPASS < AH < ESP w/integrity, but it gets complicated if we have multiple ESP encryption or integrity algorithm options.) So, one has to impose a total ordering on these security parameters to make this work, but this can be done locally.

However, this conservative strategy has a possible performance downside. If most traffic traversing an IPsec implementation for a given S/D address pair (and specified protocol) is bypassed, then a fragment-only SA for that address pair might cause a dramatic increase in the volume of traffic afforded crypto processing. If the crypto implementation cannot support high traffic rates, this could cause problems. (An IPsec implementation that is capable of line rate or near line rate crypto performance would not be adversely affected by this SA configuration approach. Nonetheless, the performance impact is a potential concern, specific to implementation capabilities.)

Another concern is that non-initial fragments sent over a dedicated SA might be used to effect overlapping reassembly attacks, when combined with an apparently acceptable initial fragment. (This sort of attack assumes creation of bogus fragments and is not a side effect of normal fragmentation.) This concern is easily addressed in

IPv4, by checking the fragment offset value to ensure that no non-initial fragments have a small enough offset to overlap port fields that should be contained in the initial fragment. Recall that the IPv4 MTU minimum is 576 bytes, and the max IP header length is 60 bytes, so any ports should be present in the initial fragment. If we require all non-initial fragments to have an offset of, say, 128 or greater, just to be on the safe side, this should prevent successful attacks of this sort. If the intent is only to protect against this sort of reassembly attack, this check need be implemented only by a receiver.

IPv6 also has a fragment offset, carried in the fragmentation extension header. However, IPv6 extension headers are variable in length and there is no analogous max header length value that we can use to check non-initial fragments, to reject ones that might be used for an attack of the sort noted above. A receiver would need to maintain state analogous to reassembly state, to provide equivalent protection. So, only for IPv4 is it feasible to impose a fragment offset check that would reject attacks designed to circumvent port field checks by IPsec (or firewalls) when passing non-initial fragments.

Another possible concern is that in some topologies and SPD configurations this approach might result in an access control surprise. The notion is that if we create an SA to carry ALL (non-initial) fragments, then that SA would carry some traffic that might otherwise arrive as plaintext via a separate path, e.g., a path monitored by a proxy firewall. But, this concern arises only if the other path allows initial fragments to traverse it without requiring reassembly, presumably a bad idea for a proxy firewall. Nonetheless, this does represent a potential problem in some topologies and under certain assumptions with respect to SPD and (other) firewall rule sets, and administrators need to be warned of this possibility.

A less serious concern is that non-initial fragments sent over a non-initial fragment-only SA might represent a DoS opportunity, in that they could be sent when no valid, initial fragment will ever arrive. This might be used to attack hosts behind an SG or BITW device. However, the incremental risk posed by this sort of attack, which can be mounted only by hosts behind an SG or BITW device, seems small.

If we interpret the ANY selector value as encompassing OPAQUE, then a single SA with ANY values for both port fields would be able to accommodate all traffic matching the S/D address and protocol traffic selectors, an alternative to using the OPAQUE value. But, using ANY

here precludes multiple, distinct SAs between the same IPsec implementations for the same address pairs and protocol. So, it is not an exactly equivalent alternative.

Fundamentally, fragment handling problems arise only when more than one SA is defined with the same S/D address and protocol selector values, but with different port field selector values.

D.4. BYPASS/DISCARD Traffic

We also have to address the non-initial fragment processing issue for BYPASS/DISCARD entries, independent of SA processing. This is largely a local matter for two reasons:

1) We have no means for coordinating SPD entries for such traffic between IPsec implementations since IKE is not invoked.
2) Many of these entries refer to traffic that is NOT directed to or received from a location that is using IPsec. So there is no peer IPsec implementation with which to coordinate via any means.

However, this document should provide guidance here, consistent with our goal of offering a well-defined, access control function for all traffic, relative to the IPsec boundary. To that end, this document says that implementations MUST support fragment reassembly for BYPASS/DISCARD traffic when port fields are specified. An implementation also MUST permit a user or administrator to accept such traffic or reject such traffic using the SPD conventions described in Section 4.4.1. The concern is that BYPASS of a cleartext, non-initial fragment arriving at an IPsec implementation could undermine the security afforded IPsec-protected traffic directed to the same destination. For example, consider an IPsec implementation configured with an SPD entry that calls for IPsec-protection of traffic between a specific source/destination address pair, and for a specific protocol and destination port, e.g., TCP traffic on port 23 (Telnet). Assume that the implementation also allows BYPASS of traffic from the same source/destination address pair and protocol, but for a different destination port, e.g., port 119 (NNTP). An attacker could send a non-initial fragment (with a forged source address) that, if bypassed, could overlap with IPsec-protected traffic from the same source and thus violate the integrity of the IPsec-protected traffic. Requiring stateful fragment checking for BYPASS entries with non-trivial port ranges prevents attacks of this sort.

D.5. Just say no to ports?

It has been suggested that we could avoid the problems described above by not allowing port field selectors to be used in tunnel mode. But the discussion above shows this to be an unnecessarily stringent approach, i.e., since no problems arise for the native OS and BITS implementations. Moreover, some WG members have described scenarios where use of tunnel mode SAs with (non-trivial) port field selectors is appropriate. So the challenge is defining a strategy that can deal with this problem in BITW and SG contexts. Also note that BYPASS/DISCARD entries in the SPD that make use of ports pose the same problems, irrespective of tunnel vs. transport mode notions.

Some folks have suggested that a firewall behind an SG or BITW should be left to enforce port-level access controls and the effects of fragmentation. However, this seems to be an incongruous suggestion in that elsewhere in IPsec (e.g., in IKE payloads) we are concerned about firewalls that always discard fragments. If many firewalls don't pass fragments in general, why should we expect them to deal with fragments in this case? So, this analysis rejects the suggestion of disallowing use of port field selectors with tunnel mode SAs.

D.6. Other Suggested Solutions

One suggestion is to reassemble fragments at the sending IPsec implementation, and thus avoid the problem entirely. This approach is invisible to a receiver and thus could be adopted as a purely local implementation option.

A more sophisticated version of this suggestion calls for establishing and maintaining minimal state from each initial fragment encountered, to allow non-initial fragments to be matched to the right SAs or SPD/cache entries. This implies an extension to the current processing model (and the old one). The IPsec implementation would intercept all fragments; capture Source/Destination IP addresses, protocol, packet ID, and port fields from initial fragments; and then use this data to map non-initial fragments to SAs that require port fields. If this approach is employed, the receiver needs to employ an equivalent scheme, as it too must verify that received fragments are consistent with SA selector values. A non-initial fragment that arrives prior to an initial fragment could be cached or discarded, awaiting arrival of the corresponding initial fragment.

A downside of both approaches noted above is that they will not always work. When a BITW device or SG is configured in a topology that might allow some fragments for a packet to be processed at different SGs or BITW devices, then there is no guarantee that all

fragments will ever arrive at the same IPsec device. This approach also raises possible processing problems. If the sender caches non-initial fragments until the corresponding initial fragment arrives, buffering problems might arise, especially at high speeds. If the non-initial fragments are discarded rather than cached, there is no guarantee that traffic will ever pass, e.g., retransmission will result in different packet IDs that cannot be matched with prior transmissions. In any case, housekeeping procedures will be needed to decide when to delete the fragment state data, adding some complexity to the system. Nonetheless, this is a viable solution in some topologies, and these are likely to be common topologies.

The Working Group rejected an earlier version of the convention of creating an SA to carry only non-initial fragments, something that was supported implicitly under the RFC 2401 model via use of OPAQUE port fields, but never clearly articulated in RFC 2401. The (rejected) text called for each non-initial fragment to be treated as protocol 44 (the IPv6 fragment header protocol ID) by the sender and receiver. This approach has the potential to make IPv4 and IPv6 fragment handling more uniform, but it does not fundamentally change the problem, nor does it address the issue of fragment handling for BYPASS/DISCARD traffic. Given the fragment overlap attack problem that IPv6 poses, it does not seem that it is worth the effort to adopt this strategy.

D.7. Consistency

Earlier, the WG agreed to allow an IPsec BITS, BITW, or SG to perform fragmentation prior to IPsec processing. If this fragmentation is performed after SA lookup at the sender, there is no "mapping to the right SA" problem. But, the receiver still needs to be able to verify that the non-initial fragments are consistent with the SA via which they are received. Since the initial fragment might be lost en route, the receiver encounters all of the potential problems noted above. Thus, if we are to be consistent in our decisions, we need to say how a receiver will deal with the non-initial fragments that arrive.

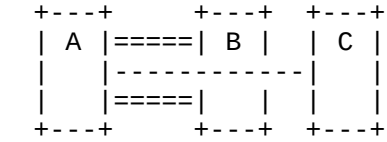
D.8. Conclusions

There is no simple, uniform way to handle fragments in all contexts. Different approaches work better in different contexts. Thus, this document offers 3 choices -- one MUST and two MAYs. At some point in the future, if the community gains experience with the two MAYs, they may become SHOULDs or MUSTs or other approaches may be proposed.

Appendix E: Example of Supporting Nested SAs via SPD and Forwarding Table Entries

This appendix provides an example of how to configure the SPD and forwarding tables to support a nested pair of SAs, consistent with the new processing model. For simplicity, this example assumes just one SPD-I.

The goal in this example is to support a transport mode SA from A to C, carried over a tunnel mode SA from A to B. For example, A might be a laptop connected to the public Internet, B might be a firewall that protects a corporate network, and C might be a server on the corporate network that demands end-to-end authentication of A's traffic.



A's SPD contains entries of the form:

Rule	Local	Remote	Next Layer Protocol	Action
1	C	A	ESP	BYPASS
2	A	C	ICMP,ESP	PROTECT(ESP,tunnel,integr+conf)
3	A	C	ANY	PROTECT(ESP,transport,integr-only)
4	A	B	ICMP,IKE	BYPASS

A's unprotected-side forwarding table is set so that outbound packets destined for C are looped back to the protected side. A's protected-side forwarding table is set so that inbound ESP packets are looped back to the unprotected side. A's forwarding tables contain entries of the form:

Unprotected-side forwarding table

Rule	Local	Remote	Protocol	Action
1	A	C	ANY	loop back to protected side
2	A	B	ANY	forward to B

Protected-side forwarding table

Rule	Local	Remote	Protocol	Action
1	A	C	ESP	loop back to unprotected side

An outbound TCP packet from A to C would match SPD rule 3 and have transport mode ESP applied to it. The unprotected-side forwarding table would then loop back the packet. The packet is compared against SPD-I (see Figure 2), matches SPD rule 1, and so it is BYPASSed. The packet is treated as an outbound packet and compared against the SPD for a third time. This time it matches SPD rule 2, so ESP is applied in tunnel mode. This time the forwarding table doesn't loop back the packet, because the outer destination address is B, so the packet goes out onto the wire.

An inbound TCP packet from C to A is wrapped in two ESP headers; the outer header (ESP in tunnel mode) shows B as the source, whereas the inner header (ESP transport mode) shows C as the source. Upon arrival at A, the packet would be mapped to an SA based on the SPI, have the outer header removed, and be decrypted and integrity-checked. Then it would be matched against the SAD selectors for this SA, which would specify C as the source and A as the destination, derived from SPD rule 2. The protected-side forwarding function would then send it back to the unprotected side based on the addresses and the next layer protocol (ESP), indicative of nesting. It is compared against SPD-O (see Figure 3) and found to match SPD rule 1, so it is BYPASSed. The packet is mapped to an SA based on the SPI, integrity-checked, and compared against the SAD selectors derived from SPD rule 3. The forwarding function then passes it up to the next layer, because it isn't an ESP packet.

References

Normative References

[BBCDWW98] Blake, S., Black, D., Carlson, M., Davies, E., Wang, Z., and W. Weiss, "An Architecture for Differentiated Service", RFC 2475, December 1998.

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

[CD98] Conta, A. and S. Deering, "Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification", RFC 2463, December 1998.

[DH98] Deering, S., and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification", RFC 2460, December 1998.

[Eas05] 3rd Eastlake, D., "Cryptographic Algorithm Implementation Requirements For Encapsulating Security Payload (ESP) and Authentication Header (AH)", RFC 4305, December 2005.

[HarCar98] Harkins, D. and D. Carrel, "The Internet Key Exchange (IKE)", RFC 2409, November 1998.

[Kau05] Kaufman, C., Ed., "The Internet Key Exchange (IKEv2) Protocol", RFC 4306, December 2005.

[Ken05a] Kent, S., "IP Encapsulating Security Payload (ESP)", RFC 4303, December 2005.

[Ken05b] Kent, S., "IP Authentication Header", RFC 4302, December 2005.

[MD90] Mogul, J. and S. Deering, "Path MTU discovery", RFC 1191, November 1990.

[Mobip] Johnson, D., Perkins, C., and J. Arkko, "Mobility Support in IPv6", RFC 3775, June 2004.

[Pos81a] Postel, J., "Internet Protocol", STD 5, RFC 791, September 1981.

[Pos81b] Postel, J., "Internet Control Message Protocol", RFC 792, September 1981.

[Sch05] Schiller, J., "Cryptographic Algorithms for use in the Internet Key Exchange Version 2 (IKEv2)", RFC 4307, December 2005.

[WaKiHo97] Wahl, M., Kille, S., and T. Howes, "Lightweight Directory Access Protocol (v3): UTF-8 String Representation of Distinguished Names", RFC 2253, December 1997.

Informative References

[CoSa04] Condell, M., and L. Sanchez, "On the Deterministic Enforcement of Un-ordered Security Policies", BBN Technical Memo 1346, March 2004.

[FaLiHaMeTr00] Farinacci, D., Li, T., Hanks, S., Meyer, D., and P. Traina, "Generic Routing Encapsulation (GRE)", RFC 2784, March 2000.

[Gro02] Grossman, D., "New Terminology and Clarifications for Diffserv", RFC 3260, April 2002.

[HC03] Holbrook, H. and B. Cain, "Source Specific Multicast for IP", Work in Progress, November 3, 2002.

[HA94] Haller, N. and R. Atkinson, "On Internet Authentication", RFC 1704, October 1994.

[NiBlBaBL98] Nichols, K., Blake, S., Baker, F., and D. Black, "Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers", RFC 2474, December 1998.

[Per96] Perkins, C., "IP Encapsulation within IP", RFC 2003, October 1996.

[RaFlBl01] Ramakrishnan, K., Floyd, S., and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP", RFC 3168, September 2001.

[RFC2401] Kent, S. and R. Atkinson, "Security Architecture for the Internet Protocol", RFC 2401, November 1998.

[RFC2983] Black, D., "Differentiated Services and Tunnels", RFC 2983, October 2000.

[RFC3547] Baugher, M., Weis, B., Hardjono, T., and H. Harney, "The Group Domain of Interpretation", RFC 3547, July 2003.

[RFC3740] Hardjono, T. and B. Weis, "The Multicast Group Security Architecture", RFC 3740, March 2004.

[RaCoCaDe04] Rajahalme, J., Conta, A., Carpenter, B., and S. Deering, "IPv6 Flow Label Specification", RFC 3697, March 2004.

[Sch94] Schneier, B., Applied Cryptography, Section 8.6, John Wiley & Sons, New York, NY, 1994.

[Shi00] Shirey, R., "Internet Security Glossary", RFC 2828, May 2000.

[SMPT01] Shacham, A., Monsour, B., Pereira, R., and M. Thomas, "IP Payload Compression Protocol (IPComp)", RFC 3173, September 2001.

[ToEgWa04] Touch, J., Eggert, L., and Y. Wang, "Use of IPsec Transport Mode for Dynamic Routing", RFC 3884, September 2004.

[VK83] V.L. Voydock & S.T. Kent, "Security Mechanisms in High-level Networks", ACM Computing Surveys, Vol. 15, No. 2, June 1983.

Authors' Addresses

Stephen Kent
BBN Technologies
10 Moulton Street
Cambridge, MA 02138
USA

Phone: +1 (617) 873-3988
EMail: kent@bbn.com

Karen Seo
BBN Technologies
10 Moulton Street
Cambridge, MA 02138
USA

Phone: +1 (617) 873-3152
EMail: kseo@bbn.com

Full Copyright Statement

Intellectual Property

Acknowledgement

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

Exhibit 5

Network Working Group
Request for Comments: 4302
Obsoletes: 2402
Category: Standards Track

S. Kent
BBN Technologies

December 2005

IP Authentication Header

Status of This Memo

Copyright Notice

Abstract

This document describes an updated version of the IP Authentication Header (AH), which is designed to provide authentication services in IPv4 and IPv6. This document obsoletes RFC 2402 (November 1998).

Table of Contents

1. Introduction

This document assumes that the reader is familiar with the terms and concepts described in the "Security Architecture for the Internet Protocol" [Ken-Arch], hereafter referred to as the Security Architecture document. In particular, the reader should be familiar with the definitions of security services offered by the Encapsulating Security Payload (ESP) [Ken-ESP] and the IP Authentication Header (AH), the concept of Security Associations, the ways in which ESP can be used in conjunction with the Authentication Header (AH), and the different key management options available for ESP and AH.

The keywords MUST, MUST NOT, REQUIRED, SHALL, SHALL NOT, SHOULD, SHOULD NOT, RECOMMENDED, MAY, and OPTIONAL, when they appear in this document, are to be interpreted as described in RFC 2119 [Bra97].

The IP Authentication Header (AH) is used to provide connectionless integrity and data origin authentication for IP datagrams (hereafter referred to as just "integrity") and to provide protection against replays. This latter, optional service may be selected, by the receiver, when a Security Association (SA) is established. (The protocol default requires the sender to increment the sequence number used for anti-replay, but the service is effective only if the receiver checks the sequence number.) However, to make use of the Extended Sequence Number feature in an interoperable fashion, AH does impose a requirement on SA management protocols to be able to negotiate this new feature (see Section 2.5.1 below).

AH provides authentication for as much of the IP header as possible, as well as for next level protocol data. However, some IP header fields may change in transit and the value of these fields, when the packet arrives at the receiver, may not be predictable by the sender. The values of such fields cannot be protected by AH. Thus, the protection provided to the IP header by AH is piecemeal. (See Appendix A.)

AH may be applied alone, in combination with the IP Encapsulating Security Payload (ESP) [Ken-ESP], or in a nested fashion (see Security Architecture document [Ken-Arch]). Security services can be provided between a pair of communicating hosts, between a pair of communicating security gateways, or between a security gateway and a host. ESP may be used to provide the same anti-replay and similar integrity services, and it also provides a confidentiality (encryption) service. The primary difference between the integrity provided by ESP and AH is the extent of the coverage. Specifically, ESP does not protect any IP header fields unless those fields are

encapsulated by ESP (e.g., via use of tunnel mode). For more details on how to use AH and ESP in various network environments, see the Security Architecture document [Ken-Arch].

Section 7 provides a brief review of the differences between this document and RFC 2402 [RFC2402].

2. Authentication Header Format

The protocol header (IPv4, IPv6, or IPv6 Extension) immediately preceding the AH header SHALL contain the value 51 in its Protocol (IPv4) or Next Header (IPv6, Extension) fields [DH98]. Figure 1 illustrates the format for AH.

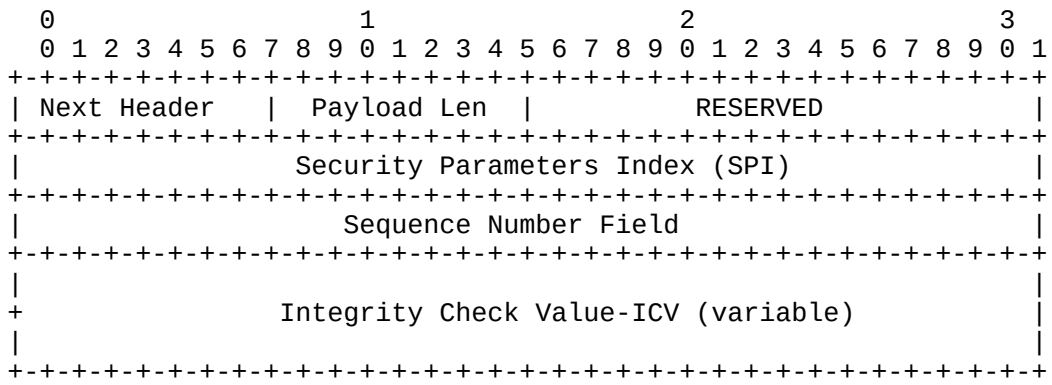


Figure 1. AH Format

The following table refers to the fields that comprise AH, (illustrated in Figure 1), plus other fields included in the integrity computation, and illustrates which fields are covered by the ICV and what is transmitted.

	# of bytes	Requ'd [1]	What Integ Covers	What is Xmtd
IP Header	variable	M	[2]	plain
Next Header	1	M	Y	plain
Payload Len	1	M	Y	plain
RESERVED	2	M	Y	plain
SPI	4	M	Y	plain
Seq# (low-order 32 bits)	4	M	Y	plain
ICV	variable	M	Y[3]	plain
IP datagram [4]	variable	M	Y	plain
Seq# (high-order 32 bits)	4	if ESN	Y	not xmtd
ICV Padding	variable	if need	Y	not xmtd

[1] - M = mandatory
[2] - See Section 3.3.3, "Integrity Check Value Calculation", for details of which IP header fields are covered.
[3] - Zeroed before ICV calculation (resulting ICV placed here after calculation)
[4] - If tunnel mode -> IP datagram
If transport mode -> next header and data

The following subsections define the fields that comprise the AH format. All the fields described here are mandatory; i.e., they are always present in the AH format and are included in the Integrity Check Value (ICV) computation (see Sections 2.6 and 3.3.3).

Note: All of the cryptographic algorithms used in IPsec expect their input in canonical network byte order (see Appendix of RFC 791 [RFC791]) and generate their output in canonical network byte order. IP packets are also transmitted in network byte order.

AH does not contain a version number, therefore if there are concerns about backward compatibility, they MUST be addressed by using a signaling mechanism between the two IPsec peers to ensure compatible versions of AH, e.g., IKE [IKEv2] or an out-of-band configuration mechanism.

2.1. Next Header

The Next Header is an 8-bit field that identifies the type of the next payload after the Authentication Header. The value of this field is chosen from the set of IP Protocol Numbers defined on the web page of Internet Assigned Numbers Authority (IANA). For example, a value of 4 indicates IPv4, a value of 41 indicates IPv6, and a value of 6 indicates TCP.

2.2. Payload Length

This 8-bit field specifies the length of AH in 32-bit words (4-byte units), minus "2". Thus, for example, if an integrity algorithm yields a 96-bit authentication value, this length field will be "4" (3 32-bit word fixed fields plus 3 32-bit words for the ICV, minus 2). For IPv6, the total length of the header must be a multiple of 8-octet units. (Note that although IPv6 [DH98] characterizes AH as an extension header, its length is measured in 32-bit words, not the 64-bit words used by other IPv6 extension headers.) See Section 2.6, "Integrity Check Value (ICV)", for comments on padding of this field, and Section 3.3.3.2.1, "ICV Padding".

2.3. Reserved

This 16-bit field is reserved for future use. It MUST be set to "zero" by the sender, and it SHOULD be ignored by the recipient. (Note that the value is included in the ICV calculation, but is otherwise ignored by the recipient.)

2.4. Security Parameters Index (SPI)

The SPI is an arbitrary 32-bit value that is used by a receiver to identify the SA to which an incoming packet is bound. For a unicast SA, the SPI can be used by itself to specify an SA, or it may be used in conjunction with the IPsec protocol type (in this case AH). Because for unicast SAs the SPI value is generated by the receiver, whether the value is sufficient to identify an SA by itself or whether it must be used in conjunction with the IPsec protocol value is a local matter. The SPI field is mandatory, and this mechanism for mapping inbound traffic to unicast SAs described above MUST be supported by all AH implementations.

If an IPsec implementation supports multicast, then it MUST support multicast SAs using the algorithm below for mapping inbound IPsec datagrams to SAs. Implementations that support only unicast traffic need not implement this de-multiplexing algorithm.

In many secure multicast architectures, e.g., [RFC3740], a central Group Controller/Key Server unilaterally assigns the group security association's SPI. This SPI assignment is not negotiated or coordinated with the key management (e.g., IKE) subsystems that reside in the individual end systems that comprise the group. Consequently, it is possible that a group security association and a unicast security association can simultaneously use the same SPI. A multicast-capable IPsec implementation MUST correctly de-multiplex inbound traffic even in the context of SPI collisions.

Each entry in the Security Association Database (SAD) [Ken-Arch] must indicate whether the SA lookup makes use of the destination, or destination and source, IP addresses, in addition to the SPI. For multicast SAs, the protocol field is not employed for SA lookups. For each inbound, IPsec-protected packet, an implementation must conduct its search of the SAD such that it finds the entry that matches the "longest" SA identifier. In this context, if two or more SAD entries match based on the SPI value, then the entry that also matches based on destination, or destination and source, address comparison (as indicated in the SAD entry) is the "longest" match. This implies a logical ordering of the SAD search as follows:

1. Search the SAD for a match on {SPI, destination address, source address}. If an SAD entry matches, then process the inbound AH packet with that matching SAD entry. Otherwise, proceed to step 2.

2. Search the SAD for a match on {SPI, destination address}. If an SAD entry matches, then process the inbound AH packet with that matching SAD entry. Otherwise, proceed to step 3.

3. Search the SAD for a match on only {SPI} if the receiver has chosen to maintain a single SPI space for AH and ESP, or on {SPI, protocol} otherwise. If an SAD entry matches, then process the inbound AH packet with that matching SAD entry. Otherwise, discard the packet and log an auditable event.

In practice, an implementation MAY choose any method to accelerate this search, although its externally visible behavior MUST be functionally equivalent to having searched the SAD in the above order. For example, a software-based implementation could index into a hash table by the SPI. The SAD entries in each hash table bucket's linked list are kept sorted to have those SAD entries with the longest SA identifiers first in that linked list. Those SAD entries having the shortest SA identifiers are sorted so that they are the last entries in the linked list. A hardware-based implementation may be able to effect the longest match search intrinsically, using commonly available Ternary Content-Addressable Memory (TCAM) features.

The indication of whether source and destination address matching is required to map inbound IPsec traffic to SAs MUST be set either as a side effect of manual SA configuration or via negotiation using an SA management protocol, e.g., IKE or Group Domain of Interpretation (GDOI) [RFC3547]. Typically, Source-Specific Multicast (SSM) [HC03] groups use a 3-tuple SA identifier composed of an SPI, a destination multicast address, and source address. An Any-Source Multicast group SA requires only an SPI and a destination multicast address as an identifier.

The set of SPI values in the range 1 through 255 is reserved by the Internet Assigned Numbers Authority (IANA) for future use; a reserved SPI value will not normally be assigned by IANA unless the use of the assigned SPI value is specified in an RFC. The SPI value of zero (0) is reserved for local, implementation-specific use and MUST NOT be sent on the wire. (For example, a key management implementation might use the zero SPI value to mean "No Security Association Exists"

during the period when the IPsec implementation has requested that its key management entity establish a new SA, but the SA has not yet been established.)

2.5. Sequence Number

This unsigned 32-bit field contains a counter value that increases by one for each packet sent, i.e., a per-SA packet sequence number. For a unicast SA or a single-sender multicast SA, the sender MUST increment this field for every transmitted packet. Sharing an SA among multiple senders is permitted, though generally not recommended. AH provides no means of synchronizing packet counters among multiple senders or meaningfully managing a receiver packet counter and window in the context of multiple senders. Thus, for a multi-sender SA, the anti-reply features of AH are not available (see Sections 3.3.2 and 3.4.3).

The field is mandatory and MUST always be present even if the receiver does not elect to enable the anti-replay service for a specific SA. Processing of the Sequence Number field is at the discretion of the receiver, but all AH implementations MUST be capable of performing the processing described in Section 3.3.2, "Sequence Number Generation", and Section 3.4.3, "Sequence Number Verification". Thus, the sender MUST always transmit this field, but the receiver need not act upon it.

The sender’s counter and the receiver’s counter are initialized to 0 when an SA is established. (The first packet sent using a given SA will have a sequence number of 1; see Section 3.3.2 for more details on how the sequence number is generated.) If anti-replay is enabled (the default), the transmitted sequence number must never be allowed to cycle. Thus, the sender’s counter and the receiver’s counter MUST be reset (by establishing a new SA and thus a new key) prior to the transmission of the 2^32nd packet on an SA.

2.5.1. Extended (64-bit) Sequence Number

To support high-speed IPsec implementations, a new option for sequence numbers SHOULD be offered, as an extension to the current, 32-bit sequence number field. Use of an Extended Sequence Number (ESN) MUST be negotiated by an SA management protocol. Note that in IKEv2, this negotiation is implicit; the default is ESN unless 32-bit sequence numbers are explicitly negotiated. (The ESN feature is applicable to multicast as well as unicast SAs.)

The ESN facility allows use of a 64-bit sequence number for an SA. (See Appendix B, "Extended (64-bit) Sequence Numbers", for details.) Only the low-order 32 bits of the sequence number are transmitted in

the AH header of each packet, thus minimizing packet overhead. The high-order 32 bits are maintained as part of the sequence number counter by both transmitter and receiver and are included in the computation of the ICV, but are not transmitted.

2.6. Integrity Check Value (ICV)

This is a variable-length field that contains the Integrity Check Value (ICV) for this packet. The field must be an integral multiple of 32 bits (IPv4 or IPv6) in length. The details of ICV processing are described in Section 3.3.3, "Integrity Check Value Calculation", and Section 3.4.4, "Integrity Check Value Verification". This field may include explicit padding, if required to ensure that the length of the AH header is an integral multiple of 32 bits (IPv4) or 64 bits (IPv6). All implementations MUST support such padding and MUST insert only enough padding to satisfy the IPv4/IPv6 alignment requirements. Details of how to compute the required padding length are provided below in Section 3.3.3.2, "Padding". The integrity algorithm specification MUST specify the length of the ICV and the comparison rules and processing steps for validation.

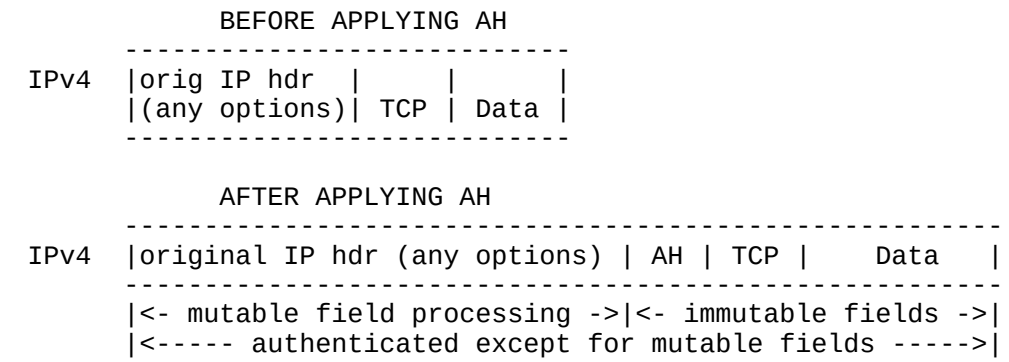
3. Authentication Header Processing

3.1. Authentication Header Location

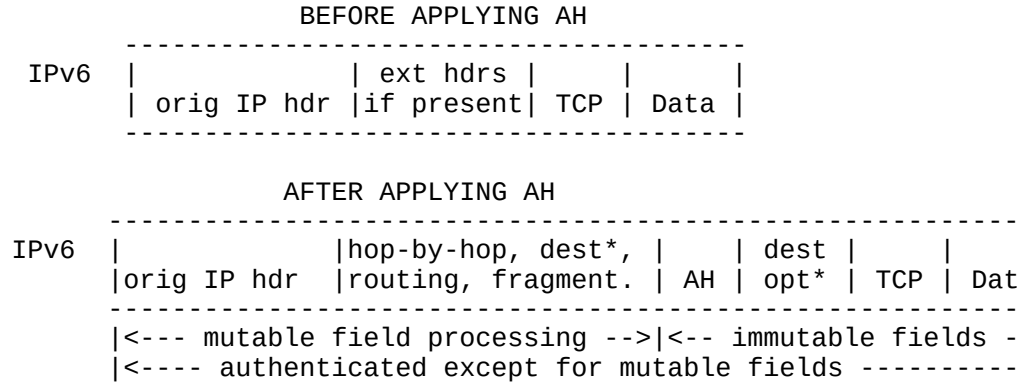
AH may be employed in two ways: transport mode or tunnel mode. (See the Security Architecture document for a description of when each should be used.)

3.1.1. Transport Mode

In transport mode, AH is inserted after the IP header and before a next layer protocol (e.g., TCP, UDP, ICMP, etc.) or before any other IPsec headers that have already been inserted. In the context of IPv4, this calls for placing AH after the IP header (and any options that it contains), but before the next layer protocol. (Note that the term "transport" mode should not be misconstrued as restricting its use to TCP and UDP.) The following diagram illustrates AH transport mode positioning for a typical IPv4 packet, on a "before and after" basis.



In the IPv6 context, AH is viewed as an end-to-end payload, and thus should appear after hop-by-hop, routing, and fragmentation extension headers. The destination options extension header(s) could appear before or after or both before and after the AH header depending on the semantics desired. The following diagram illustrates AH transport mode positioning for a typical IPv6 packet.



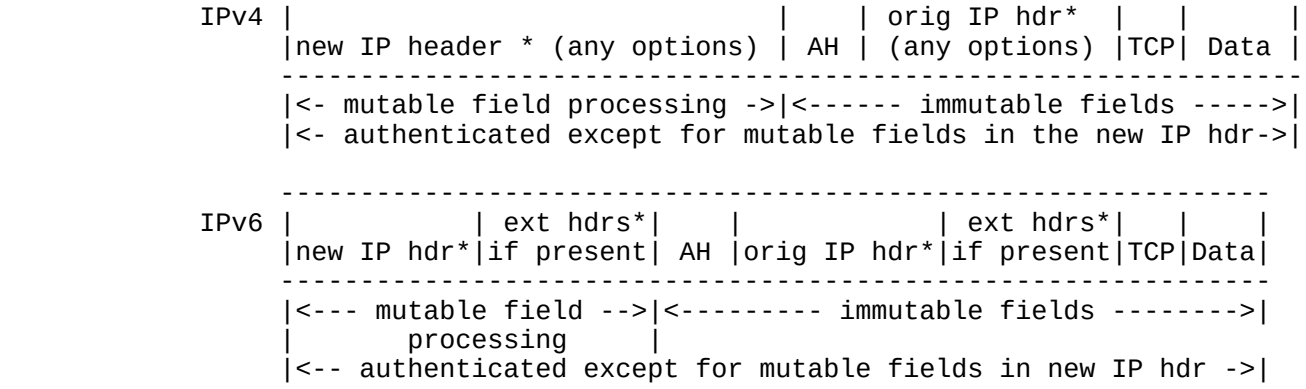
* = if present, could be before AH, after AH, or both

ESP and AH headers can be combined in a variety of modes. The IPsec Architecture document describes the combinations of security associations that must be supported.

Note that in transport mode, for "bump-in-the-stack" or "bump-in-the-wire" implementations, as defined in the Security Architecture document, inbound and outbound IP fragments may require an IPsec implementation to perform extra IP reassembly/fragmentation in order to both conform to this specification and provide transparent IPsec support. Special care is required to perform such operations within these implementations when multiple interfaces are in use.

3.1.2. Tunnel Mode

In tunnel mode, the "inner" IP header carries the ultimate (IP) source and destination addresses, while an "outer" IP header contains the addresses of the IPsec "peers," e.g., addresses of security gateways. Mixed inner and outer IP versions are allowed, i.e., IPv6 over IPv4 and IPv4 over IPv6. In tunnel mode, AH protects the entire inner IP packet, including the entire inner IP header. The position of AH in tunnel mode, relative to the outer IP header, is the same as for AH in transport mode. The following diagram illustrates AH tunnel mode positioning for typical IPv4 and IPv6 packets.



3.2. Integrity Algorithms

The integrity algorithm employed for the ICV computation is specified by the SA. For point-to-point communication, suitable integrity algorithms include keyed Message Authentication Codes (MACs) based on symmetric encryption algorithms (e.g., AES [AES]) or on one-way hash functions (e.g., MD5, SHA-1, SHA-256, etc.). For multicast communication, a variety of cryptographic strategies for providing integrity have been developed and research continues in this area.

3.3. Outbound Packet Processing

In transport mode, the sender inserts the AH header after the IP header and before a next layer protocol header, as described above. In tunnel mode, the outer and inner IP header/extensions can be

interrelated in a variety of ways. The construction of the outer IP header/extensions during the encapsulation process is described in the Security Architecture document.

3.3.1. Security Association Lookup

AH is applied to an outbound packet only after an IPsec implementation determines that the packet is associated with an SA that calls for AH processing. The process of determining what, if any, IPsec processing is applied to outbound traffic is described in the Security Architecture document.

3.3.2. Sequence Number Generation

The sender's counter is initialized to 0 when an SA is established. The sender increments the sequence number (or ESN) counter for this SA and inserts the low-order 32 bits of the value into the Sequence Number field. Thus, the first packet sent using a given SA will contain a sequence number of 1.

If anti-replay is enabled (the default), the sender checks to ensure that the counter has not cycled before inserting the new value in the Sequence Number field. In other words, the sender MUST NOT send a packet on an SA if doing so would cause the sequence number to cycle. An attempt to transmit a packet that would result in sequence number overflow is an auditable event. The audit log entry for this event SHOULD include the SPI value, current date/time, Source Address, Destination Address, and (in IPv6) the cleartext Flow ID.

The sender assumes anti-replay is enabled as a default, unless otherwise notified by the receiver (see Section 3.4.3) or if the SA was configured using manual key management. Thus, typical behavior of an AH implementation calls for the sender to establish a new SA when the Sequence Number (or ESN) cycles, or in anticipation of this value cycling.

If anti-replay is disabled (as noted above), the sender does not need to monitor or reset the counter, e.g., in the case of manual key management (see Section 5). However, the sender still increments the counter and when it reaches the maximum value, the counter rolls over back to zero. (This behavior is recommended for multi-sender, multicast SAs, unless anti-replay mechanisms outside the scope of this standard are negotiated between the sender and receiver.)

If ESN (see Appendix B) is selected, only the low-order 32 bits of the sequence number are transmitted in the Sequence Number field, although both sender and receiver maintain full 64-bit ESN counters. However, the high-order 32 bits are included in the ICV calculation.

Note: If a receiver chooses not to enable anti-replay for an SA, then the receiver SHOULD NOT negotiate ESN in an SA management protocol. Use of ESN creates a need for the receiver to manage the anti-replay window (in order to determine the correct value for the high-order bits of the ESN, which are employed in the ICV computation), which is generally contrary to the notion of disabling anti-replay for an SA.

3.3.3. Integrity Check Value Calculation

The AH ICV is computed over:

- o IP or extension header fields before the AH header that are either immutable in transit or that are predictable in value upon arrival at the endpoint for the AH SA
- o the AH header (Next Header, Payload Len, Reserved, SPI, Sequence Number (low-order 32 bits), and the ICV (which is set to zero for this computation), and explicit padding bytes (if any))
- o everything after AH is assumed to be immutable in transit
- o the high-order bits of the ESN (if employed), and any implicit padding required by the integrity algorithm

3.3.3.1. Handling Mutable Fields

If a field may be modified during transit, the value of the field is set to zero for purposes of the ICV computation. If a field is mutable, but its value at the (IPsec) receiver is predictable, then that value is inserted into the field for purposes of the ICV calculation. The Integrity Check Value field is also set to zero in preparation for this computation. Note that by replacing each field's value with zero, rather than omitting the field, alignment is preserved for the ICV calculation. Also, the zero-fill approach ensures that the length of the fields that are so handled cannot be changed during transit, even though their contents are not explicitly covered by the ICV.

As a new extension header or IPv4 option is created, it will be defined in its own RFC and SHOULD include (in the Security Considerations section) directions for how it should be handled when calculating the AH ICV. If the IP (v4 or v6) implementation encounters an extension header that it does not recognize, it will discard the packet and send an ICMP message. IPsec will never see the packet. If the IPsec implementation encounters an IPv4 option that it does not recognize, it should zero the whole option, using the second byte of the option as the length. IPv6 options (in Destination Extension Headers or the Hop-by-Hop Extension Header) contain a flag indicating mutability, which determines appropriate processing for such options.

3.3.3.1.1. ICV Computation for IPv4

3.3.3.1.1.1. Base Header Fields

The IPv4 base header fields are classified as follows:

Immutable
- Version
- Internet Header Length
- Total Length
- Identification
- Protocol (This should be the value for AH.)
- Source Address
- Destination Address (without loose or strict source routing)

Mutable but predictable
- Destination Address (with loose or strict source routing)

Mutable (zeroed prior to ICV calculation)
- Differentiated Services Code Point (DSCP)
  (6 bits, see RFC 2474 [NBBB98])
- Explicit Congestion Notification (ECN)
  (2 bits, see RFC 3168 [RFB01])
- Flags
- Fragment Offset
- Time to Live (TTL)
- Header Checksum

DSCP - Routers may rewrite the DS field as needed to provide a desired local or end-to-end service, thus its value upon reception cannot be predicted by the sender.

ECN - This will change if a router along the route experiences congestion, and thus its value upon reception cannot be predicted by the sender.

Flags - This field is excluded because an intermediate router might set the DF bit, even if the source did not select it.

Fragment Offset - Since AH is applied only to non-fragmented IP packets, the Offset Field must always be zero, and thus it is excluded (even though it is predictable).

TTL - This is changed en route as a normal course of processing by routers, and thus its value at the receiver is not predictable by the sender.

Header Checksum - This will change if any of these other fields change, and thus its value upon reception cannot be predicted by the sender.

3.3.3.1.1.2. Options

For IPv4 (unlike IPv6), there is no mechanism for tagging options as mutable in transit. Hence the IPv4 options are explicitly listed in Appendix A and classified as immutable, mutable but predictable, or mutable. For IPv4, the entire option is viewed as a unit; so even though the type and length fields within most options are immutable in transit, if an option is classified as mutable, the entire option is zeroed for ICV computation purposes.

3.3.3.1.2. ICV Computation for IPv6

3.3.3.1.2.1. Base Header Fields

The IPv6 base header fields are classified as follows:

Immutable
- Version
- Payload Length
- Next Header
- Source Address
- Destination Address (without Routing Extension Header)

Mutable but predictable
- Destination Address (with Routing Extension Header)

Mutable (zeroed prior to ICV calculation)
- DSCP (6 bits, see RFC2474 [NBBB98])
- ECN (2 bits, see RFC3168 [RFB01])
- Flow Label (*)
- Hop Limit

(*) The flow label described in AHv1 was mutable, and in RFC 2460 [DH98] was potentially mutable. To retain compatibility with existing AH implementations, the flow label is not included in the ICV in AHv2.

3.3.3.1.2.2. Extension Headers Containing Options

IPv6 options in the Hop-by-Hop and Destination Extension Headers contain a bit that indicates whether the option might change (unpredictably) during transit. For any option for which contents may change en-route, the entire "Option Data" field must be treated as zero-valued octets when computing or verifying the ICV. The

Option Type and Opt Data Len are included in the ICV calculation. All options for which the bit indicates immutability are included in the ICV calculation. See the IPv6 specification [DH98] for more information.

3.3.3.1.2.3. Extension Headers Not Containing Options

The IPv6 extension headers that do not contain options are explicitly listed in Appendix A and classified as immutable, mutable but predictable, or mutable.

3.3.3.2. Padding and Extended Sequence Numbers

3.3.3.2.1. ICV Padding

As mentioned in Section 2.6, the ICV field may include explicit padding if required to ensure that the AH header is a multiple of 32 bits (IPv4) or 64 bits (IPv6). If padding is required, its length is determined by two factors:

- the length of the ICV
- the IP protocol version (v4 or v6)

For example, if the output of the selected algorithm is 96 bits, no padding is required for IPv4 or IPv6. However, if a different length ICV is generated, due to use of a different algorithm, then padding may be required depending on the length and IP protocol version. The content of the padding field is arbitrarily selected by the sender. (The padding is arbitrary, but need not be random to achieve security.) These padding bytes are included in the ICV calculation, counted as part of the Payload Length, and transmitted at the end of the ICV field to enable the receiver to perform the ICV calculation. Inclusion of padding in excess of the minimum amount required to satisfy IPv4/IPv6 alignment requirements is prohibited.

3.3.3.2.2. Implicit Packet Padding and ESN

If the ESN option is elected for an SA, then the high-order 32 bits of the ESN must be included in the ICV computation. For purposes of ICV computation, these bits are appended (implicitly) immediately after the end of the payload, and before any implicit packet padding.

For some integrity algorithms, the byte string over which the ICV computation is performed must be a multiple of a blocksize specified by the algorithm. If the IP packet length (including AH and the 32 high-order bits of the ESN, if enabled) does not match the blocksize requirements for the algorithm, implicit padding MUST be appended to the end of the packet, prior to ICV computation. The padding octets

MUST have a value of zero. The blocksize (and hence the length of the padding) is specified by the algorithm specification. This padding is not transmitted with the packet. The document that defines an integrity algorithm MUST be consulted to determine if implicit padding is required as described above. If the document does not specify an answer to this, then the default is to assume that implicit padding is required (as needed to match the packet length to the algorithm's blocksize.) If padding bytes are needed but the algorithm does not specify the padding contents, then the padding octets MUST have a value of zero.

3.3.4. Fragmentation

If required, IP fragmentation occurs after AH processing within an IPsec implementation. Thus, transport mode AH is applied only to whole IP datagrams (not to IP fragments). An IPv4 packet to which AH has been applied may itself be fragmented by routers en route, and such fragments must be reassembled prior to AH processing at a receiver. (This does not apply to IPv6, where there is no router-initiated fragmentation.) In tunnel mode, AH is applied to an IP packet, the payload of which may be a fragmented IP packet. For example, a security gateway or a "bump-in-the-stack" or "bump-in-the-wire" IPsec implementation (see the Security Architecture document for details) may apply tunnel mode AH to such fragments.

NOTE: For transport mode -- As mentioned at the end of Section 3.1.1, bump-in-the-stack and bump-in-the-wire implementations may have to first reassemble a packet fragmented by the local IP layer, then apply IPsec, and then fragment the resulting packet.

NOTE: For IPv6 -- For bump-in-the-stack and bump-in-the-wire implementations, it will be necessary to examine all the extension headers to determine if there is a fragmentation header and hence that the packet needs reassembling prior to IPsec processing.

Fragmentation, whether performed by an IPsec implementation or by routers along the path between IPsec peers, significantly reduces performance. Moreover, the requirement for an AH receiver to accept fragments for reassembly creates denial of service vulnerabilities. Thus, an AH implementation MAY choose to not support fragmentation and may mark transmitted packets with the DF bit, to facilitate Path MTU (PMTU) discovery. In any case, an AH implementation MUST support generation of ICMP PMTU messages (or equivalent internal signaling for native host implementations) to minimize the likelihood of fragmentation. Details of the support required for MTU management are contained in the Security Architecture document.

3.4. Inbound Packet Processing

If there is more than one IPsec header/extension present, the processing for each one ignores (does not zero, does not use) any IPsec headers applied subsequent to the header being processed.

3.4.1. Reassembly

If required, reassembly is performed prior to AH processing. If a packet offered to AH for processing appears to be an IP fragment, i.e., the OFFSET field is nonzero or the MORE FRAGMENTS flag is set, the receiver MUST discard the packet; this is an auditable event. The audit log entry for this event SHOULD include the SPI value, date/time, Source Address, Destination Address, and (in IPv6) the Flow ID.

NOTE: For packet reassembly, the current IPv4 spec does NOT require either the zeroing of the OFFSET field or the clearing of the MORE FRAGMENTS flag. In order for a reassembled packet to be processed by IPsec (as opposed to discarded as an apparent fragment), the IP code must do these two things after it reassembles a packet.

3.4.2. Security Association Lookup

Upon receipt of a packet containing an IP Authentication Header, the receiver determines the appropriate (unidirectional) SA via lookup in the SAD. For a unicast SA, this determination is based on the SPI or the SPI plus protocol field, as described in Section 2.4. If an implementation supports multicast traffic, the destination address is also employed in the lookup (in addition to the SPI), and the sender address also may be employed, as described in Section 2.4. (This process is described in more detail in the Security Architecture document.) The SAD entry for the SA also indicates whether the Sequence Number field will be checked and whether 32- or 64-bit sequence numbers are employed for the SA. The SAD entry for the SA also specifies the algorithm(s) employed for ICV computation, and indicates the key required to validate the ICV.

If no valid Security Association exists for this packet the receiver MUST discard the packet; this is an auditable event. The audit log entry for this event SHOULD include the SPI value, date/time, Source Address, Destination Address, and (in IPv6) the Flow ID.

(Note that SA management traffic, such as IKE packets, does not need to be processed based on SPI, i.e., one can de-multiplex this traffic separately based on Next Protocol and Port fields, for example.)

3.4.3. Sequence Number Verification

All AH implementations MUST support the anti-replay service, though its use may be enabled or disabled by the receiver on a per-SA basis. Anti-replay is applicable to unicast as well as multicast SAs. However, this standard specifies no mechanisms for providing anti-replay for a multi-sender SA (unicast or multicast). In the absence of negotiation (or manual configuration) of an anti-replay mechanism for such an SA, it is recommended that sender and receiver checking of the Sequence Number for the SA be disabled (via negotiation or manual configuration), as noted below.

If the receiver does not enable anti-replay for an SA, no inbound checks are performed on the Sequence Number. However, from the perspective of the sender, the default is to assume that anti-replay is enabled at the receiver. To avoid having the sender do unnecessary sequence number monitoring and SA setup (see Section 3.3.2, "Sequence Number Generation"), if an SA establishment protocol such as IKE is employed, the receiver SHOULD notify the sender, during SA establishment, if the receiver will not provide anti-replay protection.

If the receiver has enabled the anti-replay service for this SA, the receive packet counter for the SA MUST be initialized to zero when the SA is established. For each received packet, the receiver MUST verify that the packet contains a Sequence Number that does not duplicate the Sequence Number of any other packets received during the life of this SA. This SHOULD be the first AH check applied to a packet after it has been matched to an SA, to speed rejection of duplicate packets.

Duplicates are rejected through the use of a sliding receive window. How the window is implemented is a local matter, but the following text describes the functionality that the implementation must exhibit.

The "right" edge of the window represents the highest, validated Sequence Number value received on this SA. Packets that contain sequence numbers lower than the "left" edge of the window are rejected. Packets falling within the window are checked against a list of received packets within the window.

If the ESN option is selected for an SA, only the low-order 32 bits of the sequence number are explicitly transmitted, but the receiver employs the full sequence number computed using the high-order 32 bits for the indicated SA (from his local counter) when checking the received Sequence Number against the receive window. In constructing the full sequence number, if the low-order 32 bits carried in the

packet are lower in value than the low-order 32 bits of the receiver's sequence number counter, the receiver assumes that the high-order 32 bits have been incremented, moving to a new sequence number subspace. (This algorithm accommodates gaps in reception for a single SA as large as 2**32-1 packets. If a larger gap occurs, additional, heuristic checks for re-synchronization of the receiver's sequence number counter MAY be employed, as described in Appendix B.)

If the received packet falls within the window and is not a duplicate, or if the packet is to the right of the window, then the receiver proceeds to ICV verification. If the ICV validation fails, the receiver MUST discard the received IP datagram as invalid. This is an auditable event. The audit log entry for this event SHOULD include the SPI value, date/time, Source Address, Destination Address, the Sequence Number, and (in IPv6) the Flow ID. The receive window is updated only if the ICV verification succeeds.

A MINIMUM window size of 32 packets MUST be supported, but a window size of 64 is preferred and SHOULD be employed as the default. Another window size (larger than the MINIMUM) MAY be chosen by the receiver. (The receiver does NOT notify the sender of the window size.) The receive window size should be increased for higher-speed environments, irrespective of assurance issues. Values for minimum and recommended receive window sizes for very high-speed (e.g., multi-gigabit/second) devices are not specified by this standard.

3.4.4. Integrity Check Value Verification

The receiver computes the ICV over the appropriate fields of the packet, using the specified integrity algorithm, and verifies that it is the same as the ICV included in the ICV field of the packet. Details of the computation are provided below.

If the computed and received ICVs match, then the datagram is valid, and it is accepted. If the test fails, then the receiver MUST discard the received IP datagram as invalid. This is an auditable event. The audit log entry SHOULD include the SPI value, date/time received, Source Address, Destination Address, and (in IPv6) the Flow ID.

Implementation Note:

> Implementations can use any set of steps that results in the same result as the following set of steps. Begin by saving the ICV value and replacing it (but not any ICV field padding) with zero. Zero all other fields that may have been modified during transit. (See Section 3.3.3.1, "Handling Mutable Fields", for a discussion of which fields are zeroed before performing the ICV calculation.)

If the ESN option is elected for this SA, append the high-order 32 bits of the ESN after the end of the packet. Check the overall length of the packet (as described above), and if it requires implicit padding based on the requirements of the integrity algorithm, append zero-filled bytes to the end of the packet (after the ESN if present) as required. Perform the ICV computation and compare the result with the saved value, using the comparison rules defined by the algorithm specification. (For example, if a digital signature and one-way hash are used for the ICV computation, the matching process is more complex.)

4. Auditing

Not all systems that implement AH will implement auditing. However, if AH is incorporated into a system that supports auditing, then the AH implementation MUST also support auditing and MUST allow a system administrator to enable or disable auditing for AH. For the most part, the granularity of auditing is a local matter. However, several auditable events are identified in this specification, and for each of these events a minimum set of information that SHOULD be included in an audit log is defined. Additional information also MAY be included in the audit log for each of these events, and additional events, not explicitly called out in this specification, also MAY result in audit log entries. There is no requirement for the receiver to transmit any message to the purported sender in response to the detection of an auditable event, because of the potential to induce denial of service via such action.

5. Conformance Requirements

Implementations that claim conformance or compliance with this specification MUST fully implement the AH syntax and processing described here for unicast traffic, and MUST comply with all requirements of the Security Architecture document [Ken-Arch]. Additionally, if an implementation claims to support multicast traffic, it MUST comply with the additional requirements specified for support of such traffic. If the key used to compute an ICV is manually distributed, correct provision of the anti-replay service would require correct maintenance of the counter state at the sender, until the key is replaced, and there likely would be no automated recovery provision if counter overflow were imminent. Thus, a compliant implementation SHOULD NOT provide this service in conjunction with SAs that are manually keyed.

The mandatory-to-implement algorithms for use with AH are described in a separate RFC [Eas04], to facilitate updating the algorithm requirements independently from the protocol per se. Additional algorithms, beyond those mandated for AH, MAY be supported.

6. Security Considerations

Security is central to the design of this protocol, and these security considerations permeate the specification. Additional security-relevant aspects of using the IPsec protocol are discussed in the Security Architecture document.

7. Differences from RFC 2402

This document differs from RFC 2402 [RFC2402] in the following ways.

- SPI -- modified to specify a uniform algorithm for SAD lookup for unicast and multicast SAs, covering a wider range of multicast technologies. For unicast, the SPI may be used alone to select an SA, or may be combined with the protocol, at the option of the receiver. For multicast SAs, the SPI is combined with the destination address, and optionally the source address, to select an SA.
- Extended Sequence Number -- added a new option for a 64-bit sequence number for very high-speed communications. Clarified sender and receiver processing requirements for multicast SAs and multi-sender SAs.
- Moved references to mandatory algorithms to a separate document [Eas04].

8. Acknowledgements

The author would like to acknowledge the contributions of Ran Atkinson, who played a critical role in initial IPsec activities, and who authored the first series of IPsec standards: RFCs 1825-1827. Karen Seo deserves special thanks for providing help in the editing of this and the previous version of this specification. The author also would like to thank the members of the IPsec and MSEC working groups who have contributed to the development of this protocol specification.

9. References

9.1. Normative References

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

[DH98] Deering, S. and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification", RFC 2460, December 1998.

[Eas04] 3rd Eastlake, D., "Cryptographic Algorithm Implementation Requirements for Encapsulating Security Payload (ESP) and Authentication Header (AH)", RFC 4305, December 2005.

[Ken-Arch] Kent, S. and K. Seo, "Security Architecture for the Internet Protocol", RFC 4301, December 2005.

[RFC791] Postel, J., "Internet Protocol", STD 5, RFC 791, September 1981.

[RFC1108] Kent, S., "U.S. Department of Defense Security Options for the Internet Protocol", RFC 1108, November 1991.

9.2. Informative References

[AES] Advanced Encryption Standard (AES), Federal Information Processing Standard 197, National Institutes of Standards and Technology, November 26, 2001.

[HC03] Holbrook, H. and B. Cain, "Source Specific Multicast for IP", Work in Progress, November 3, 2002.

[IKEv2] Kaufman, C., Ed., "Internet Key Exchange (IKEv2) Protocol", RFC 4306, December 2005.

[Ken-ESP] Kent, S., "IP Encapsulating Security Payload (ESP)", RFC 4303, December 2005.

[NBBB98] Nichols, K., Blake, S., Baker, F., and D. Black, "Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers", RFC 2474, December 1998.

[RFB01] Ramakrishnan, K., Floyd, S., and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP", RFC 3168, September 2001.

[RFC1063] Mogul, J., Kent, C., Partridge, C., and K. McCloghrie, "IP MTU discovery options", RFC 1063, July 1988.

[RFC1122] Braden, R., "Requirements for Internet Hosts - Communication Layers", STD 3, RFC 1122, October 1989.

[RFC1191] Mogul, J. and S. Deering, "Path MTU discovery", RFC 1191, November 1990.

[RFC1385] Wang, Z., "EIP: The Extended Internet Protocol", RFC 1385, November 1992.

[RFC1393] Malkin, G., "Traceroute Using an IP Option", RFC 1393, January 1993.

[RFC1770] Graff, C., "IPv4 Option for Sender Directed Multi-Destination Delivery", RFC 1770, March 1995.

[RFC2113] Katz, D., "IP Router Alert Option", RFC 2113, February 1997.

[RFC2402] Kent, S. and R. Atkinson, "IP Authentication Header", RFC 2402, November 1998.

[RFC3547] Baugher, M., Weis, B., Hardjono, T., and H. Harney, "The Group Domain of Interpretation", RFC 3547, July 2003.

[RFC3740] Hardjono, T. and B. Weis, "The Multicast Group Security Architecture", RFC 3740, March 2004.

Appendix A: Mutability of IP Options/Extension Headers

A1. IPv4 Options

This table shows how the IPv4 options are classified with regard to "mutability". Where two references are provided, the second one supercedes the first. This table is based in part on information provided in RFC 1700, "ASSIGNED NUMBERS", (October 1994).

Copy	Class	Opt. #	Name	Reference
IMMUTABLE -- included in ICV calculation				
0	0	0	End of Options List	[RFC791]
0	0	1	No Operation	[RFC791]
1	0	2	Security	[RFC1108] (historic but in use)
1	0	5	Extended Security	[RFC1108] (historic but in use)
1	0	6	Commercial Security	
1	0	20	Router Alert	[RFC2113]
1	0	21	Sender Directed Multi-Destination Delivery	[RFC1770]
MUTABLE -- zeroed				
1	0	3	Loose Source Route	[RFC791]
0	2	4	Time Stamp	[RFC791]
0	0	7	Record Route	[RFC791]
1	0	9	Strict Source Route	[RFC791]
0	2	18	Traceroute	[RFC1393]
EXPERIMENTAL, SUPERCEDED -- zeroed				
1	0	8	Stream ID	[RFC791, RFC1122 (Host Req)]
0	0	11	MTU Probe	[RFC1063, RFC1191 (PMTU)]
0	0	12	MTU Reply	[RFC1063, RFC1191 (PMTU)]
1	0	17	Extended Internet Protocol	[RFC1385, DH98 (IPv6)]
0	0	10	Experimental Measurement	
1	2	13	Experimental Flow Control	
1	0	14	Experimental Access Ctl	
0	0	15	???	
1	0	16	IMI Traffic Descriptor	
1	0	19	Address Extension	

NOTE: Use of the Router Alert option is potentially incompatible with use of IPsec. Although the option is immutable, its use implies that each router along a packet's path will "process" the packet and consequently might change the packet. This would happen on a hop-by-hop basis as the packet goes from router to router. Prior to

being processed by the application to which the option contents are directed (e.g., Resource Reservation Protocol (RSVP)/Internet Group Management Protocol (IGMP)), the packet should encounter AH processing. However, AH processing would require that each router along the path is a member of a multicast-SA defined by the SPI. This might pose problems for packets that are not strictly source routed, and it requires multicast support techniques not currently available.

NOTE: Addition or removal of security labels (e.g., Basic Security Option (BSO), Extended Security Option (ESO), or Commercial Internet Protocol Security Option (CIPSO)) by systems along a packet's path conflicts with the classification of these IP options as immutable and is incompatible with the use of IPsec.

NOTE: End of Options List options SHOULD be repeated as necessary to ensure that the IP header ends on a 4-byte boundary in order to ensure that there are no unspecified bytes that could be used for a covert channel.

A2. IPv6 Extension Headers

This table shows how the IPv6 extension headers are classified with regard to "mutability".

Option/Extension Name	Reference
MUTABLE BUT PREDICTABLE -- included in ICV calculation	
Routing (Type 0)	[DH98]
BIT INDICATES IF OPTION IS MUTABLE (CHANGES UNPREDICTABLY DURING TRANSIT)	
Hop-by-Hop options	[DH98]
Destination options	[DH98]
NOT APPLICABLE	
Fragmentation	[DH98]

Options -- IPv6 options in the Hop-by-Hop and Destination Extension Headers contain a bit that indicates whether the option might change (unpredictably) during transit. For any option for which contents may change en route, the entire "Option Data" field must be treated as zero-valued octets when computing or verifying the ICV. The Option Type and Opt Data Len are included in the ICV calculation. All options for which the bit indicates immutability are included in the ICV calculation. See the IPv6 specification [DH98] for more information.

Routing (Type 0) -- The IPv6 Routing Header "Type 0" will rearrange the address fields within the packet during transit from source to destination. However, the contents of the packet as it will appear at the receiver are known to the sender and to all intermediate hops. Hence, the IPv6 Routing Header "Type 0" is included in the Integrity Check Value calculation as mutable but predictable. The sender must order the field so that it appears as it will at the receiver, prior to performing the ICV computation.

Fragmentation -- Fragmentation occurs after outbound IPsec processing (Section 3.3) and reassembly occurs before inbound IPsec processing (Section 3.4). So the Fragmentation Extension Header, if it exists, is not seen by IPsec.

Note that on the receive side, the IP implementation could leave a Fragmentation Extension Header in place when it does re-assembly. If this happens, then when AH receives the packet, before doing ICV processing, AH MUST "remove" (or skip over) this header and change the previous header's "Next Header" field to be the "Next Header" field in the Fragmentation Extension Header.

Note that on the send side, the IP implementation could give the IPsec code a packet with a Fragmentation Extension Header with Offset of 0 (first fragment) and a More Fragments Flag of 0 (last fragment). If this happens, then before doing ICV processing, AH MUST first "remove" (or skip over) this header and change the previous header's "Next Header" field to be the "Next Header" field in the Fragmentation Extension Header.

Appendix B: Extended (64-bit) Sequence Numbers

B1. Overview

This appendix describes an Extended Sequence Number (ESN) scheme for use with IPsec (ESP and AH) that employs a 64-bit sequence number, but in which only the low-order 32 bits are transmitted as part of each packet. It covers both the window scheme used to detect replayed packets and the determination of the high-order bits of the sequence number that are used both for replay rejection and for computation of the ICV. It also discusses a mechanism for handling loss of synchronization relative to the (not transmitted) high-order bits.

B2. Anti-Replay Window

The receiver will maintain an anti-replay window of size W. This window will limit how far out of order a packet can be, relative to the packet with the highest sequence number that has been authenticated so far. (No requirement is established for minimum or recommended sizes for this window, beyond the 32- and 64-packet values already established for 32-bit sequence number windows. However, it is suggested that an implementer scale these values consistent with the interface speed supported by an implementation that makes use of the ESN option. Also, the algorithm described below assumes that the window is no greater than 2^31 packets in width.) All 2^32 sequence numbers associated with any fixed value for the high-order 32 bits (Seqh) will hereafter be called a sequence number subspace. The following table lists pertinent variables and their definitions.

Var. Name	Size (bits)	Meaning
W	32	Size of window
T	64	Highest sequence number authenticated so far, upper bound of window
Tl	32	Lower 32 bits of T
Th	32	Upper 32 bits of T
B	64	Lower bound of window
Bl	32	Lower 32 bits of B
Bh	32	Upper 32 bits of B
Seq	64	Sequence Number of received packet
Seql	32	Lower 32 bits of Seq
Seqh	32	Upper 32 bits of Seq

When performing the anti-replay check, or when determining which high-order bits to use to authenticate an incoming packet, there are two cases:

- Case A: Tl >= (W - 1). In this case, the window is within one sequence number subspace. (See Figure 1)
- Case B: Tl < (W - 1). In this case, the window spans two sequence number subspaces. (See Figure 2)

In the figures below, the bottom line ("----") shows two consecutive sequence number subspaces, with zeros indicating the beginning of each subspace. The two shorter lines above it show the higher-order bits that apply. The "====" represents the window. The "****" represents future sequence numbers, i.e., those beyond the current highest sequence number authenticated (ThTl).

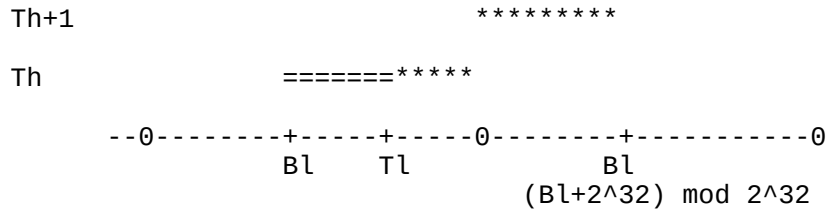


Figure 1 -- Case A

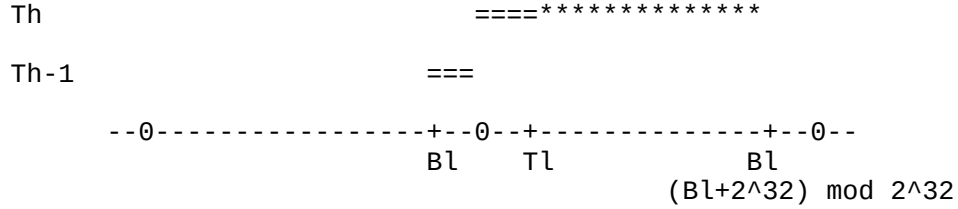


Figure 2 -- Case B

B2.1. Managing and Using the Anti-Replay Window

The anti-replay window can be thought of as a string of bits where 'W' defines the length of the string. W = T - B + 1 and cannot exceed 2^32 - 1 in value. The bottom-most bit corresponds to B and the top-most bit corresponds to T, and each sequence number from Bl through Tl is represented by a corresponding bit. The value of the bit indicates whether or not a packet with that sequence number has been received and authenticated, so that replays can be detected and rejected.

When a packet with a 64-bit sequence number (Seq) greater than T is received and validated,

+ B is increased by (Seq - T)
+ (Seq - T) bits are dropped from the low end of the window
+ (Seq - T) bits are added to the high end of the window
+ The top bit is set to indicate that a packet with that sequence number has been received and authenticated
+ The new bits between T and the top bit are set to indicate that no packets with those sequence numbers have been received yet.
+ T is set to the new sequence number

In checking for replayed packets,

+ Under Case A: If Seql >= Bl (where Bl = Tl - W + 1) AND Seql <= Tl, then check the corresponding bit in the window to see if this Seql has already been seen. If yes, reject the packet. If no, perform integrity check (see Appendix B2.2 below for determination of SeqH).

+ Under Case B: If Seql >= Bl (where Bl = Tl - W + 1) OR Seql <= Tl, then check the corresponding bit in the window to see if this Seql has already been seen. If yes, reject the packet. If no, perform integrity check (see Appendix B2.2 below for determination of Seqh).

B2.2. Determining the Higher-Order Bits (Seqh) of the Sequence Number

Because only ‘Seql’ will be transmitted with the packet, the receiver must deduce and track the sequence number subspace into which each packet falls, i.e., determine the value of Seqh. The following equations define how to select Seqh under "normal" conditions; see Appendix B3 for a discussion of how to recover from extreme packet loss.

+ Under Case A (Figure 1):
  If Seql >= Bl (where Bl = Tl - W + 1), then Seqh = Th
  If Seql < Bl (where Bl = Tl - W + 1), then Seqh = Th + 1

+ Under Case B (Figure 2):
  If Seql >= Bl (where Bl = Tl - W + 1), then Seqh = Th - 1
  If Seql < Bl (where Bl = Tl - W + 1), then Seqh = Th

B2.3. Pseudo-Code Example

The following pseudo-code illustrates the above algorithms for anti-replay and integrity checks. The values for 'Seql', 'Tl', 'Th', and 'W' are 32-bit unsigned integers. Arithmetic is mod 2^32.

```
     If (Tl >= W - 1)                              Case A
         If (Seql >= Tl - W + 1)
             Seqh = Th
             If (Seql <= Tl)
                 If (pass replay check)
                     If (pass integrity check)
                         Set bit corresponding to Seql
                         Pass the packet on
                     Else reject packet
                 Else reject packet
             Else
                 If (pass integrity check)
                     Tl = Seql (shift bits)
                     Set bit corresponding to Seql
                     Pass the packet on
                 Else reject packet
         Else
             Seqh = Th + 1
             If (pass integrity check)
                 Tl = Seql (shift bits)
                 Th = Th + 1
                 Set bit corresponding to Seql
                 Pass the packet on
             Else reject packet
     Else                                          Case B
         If (Seql >= Tl - W + 1)
             Seqh = Th - 1
             If (pass replay check)
                 If (pass integrity check)
                     Set the bit corresponding to Seql
                     Pass packet on
                 Else reject packet
             Else reject packet
         Else
             Seqh = Th
             If (Seql <= Tl)
                 If (pass replay check)
                     If (pass integrity check)
                         Set the bit corresponding to Seql
                         Pass packet on
                     Else reject packet
                 Else reject packet
```

```
            Else
                If (pass integrity check)
                    Tl = Seql (shift bits)
                    Set the bit corresponding to Seql
                    Pass packet on
                Else reject packet
```

B3. Handling Loss of Synchronization due to Significant Packet Loss

If there is an undetected packet loss of 2^32 or more consecutive packets on a single SA, then the transmitter and receiver will lose synchronization of the high-order bits, i.e., the equations in Appendix B2.2. will fail to yield the correct value. Unless this problem is detected and addressed, subsequent packets on this SA will fail authentication checks and be discarded. The following procedure SHOULD be implemented by any IPsec (ESP or AH) implementation that supports the ESN option.

Note that this sort of extended traffic loss seems unlikely to occur if any significant fraction of the traffic on the SA in question is TCP, because the source would fail to receive ACKs and would stop sending long before 2^32 packets had been lost. Also, for any bi-directional application, even ones operating above UDP, such an extended outage would likely result in triggering some form of timeout. However, a unidirectional application, operating over UDP, might lack feedback that would cause automatic detection of a loss of this magnitude, hence the motivation to develop a recovery method for this case.

The solution we've chosen was selected to:

+ minimize the impact on normal traffic processing.

+ avoid creating an opportunity for a new denial of service attack such as might occur by allowing an attacker to force diversion of resources to a re-synchronization process.
+ limit the recovery mechanism to the receiver because anti-replay is a service only for the receiver, and the transmitter generally is not aware of whether the receiver is using sequence numbers in support of this optional service. It is preferable for recovery mechanisms to be local to the receiver. This also allows for backward compatibility.

B3.1. Triggering Re-synchronization

For each SA, the receiver records the number of consecutive packets that fail authentication. This count is used to trigger the re-synchronization process, which should be performed in the background or using a separate processor. Receipt of a valid packet on the SA resets the counter to zero. The value used to trigger the re-synchronization process is a local parameter. There is no requirement to support distinct trigger values for different SAs, although an implementer may choose to do so.

B3.2. Re-synchronization Process

When the above trigger point is reached, a "bad" packet is selected for which authentication is retried using successively larger values for the upper half of the sequence number (Seqh). These values are generated by incrementing by one for each retry. The number of retries should be limited, in case this is a packet from the "past" or a bogus packet. The limit value is a local parameter. (Because the Seqh value is implicitly placed after the AH (or ESP) payload, it may be possible to optimize this procedure by executing the integrity algorithm over the packet up to the endpoint of the payload, then compute different candidate ICVs by varying the value of Seqh.) Successful authentication of a packet via this procedure resets the consecutive failure count and sets the value of T to that of the received packet.

This solution requires support only on the part of the receiver, thereby allowing for backward compatibility. Also, because re-synchronization efforts would either occur in the background or utilize an additional processor, this solution does not impact traffic processing and a denial of service attack cannot divert resources away from traffic processing.

Author’s Address

Stephen Kent
BBN Technologies
10 Moulton Street
Cambridge, MA 02138
USA

Phone: +1 (617) 873-3988
EMail: kent@bbn.com

Full Copyright Statement

Intellectual Property

Acknowledgement

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

Exhibit 6

Internet Engineering Task Force (IETF)
Request for Comments: 6101
Category: Historic
ISSN: 2070-1721

A. Freier
P. Karlton
Netscape Communications
P. Kocher
Independent Consultant

August 2011

The Secure Sockets Layer (SSL) Protocol Version 3.0

Abstract

This document is published as a historical record of the SSL 3.0 protocol. The original Abstract follows.

This document specifies version 3.0 of the Secure Sockets Layer (SSL 3.0) protocol, a security protocol that provides communications privacy over the Internet. The protocol allows client/server applications to communicate in a way that is designed to prevent eavesdropping, tampering, or message forgery.

Foreword

Although the SSL 3.0 protocol is a widely implemented protocol, a pioneer in secure communications protocols, and the basis for Transport Layer Security (TLS), it was never formally published by the IETF, except in several expired Internet-Drafts. This allowed no easy referencing to the protocol. We believe a stable reference to the original document should exist and for that reason, this document describes what is known as the last published version of the SSL 3.0 protocol, that is, the November 18, 1996, version of the protocol.

There were no changes to the original document other than trivial editorial changes and the addition of a "Security Considerations" section. However, portions of the original document that no longer apply were not included. Such as the "Patent Statement" section, the "Reserved Ports Assignment" section, and the cipher-suite registrator note in the "The CipherSuite" section. The "US export rules" discussed in the document do not apply today but are kept intact to provide context for decisions taken in protocol design. The "Goals of This Document" section indicates the goals for adopters of SSL 3.0, not goals of the IETF.

The authors and editors were retained as in the original document. The editor of this document is Nikos Mavrogiannopoulos (nikos.mavrogiannopoulos@esat.kuleuven.be). The editor would like to thank Dan Harkins, Linda Dunbar, Sean Turner, and Geoffrey Keating for reviewing this document and providing helpful comments.

Status of This Memo

This document is not an Internet Standards Track specification; it is published for the historical record.

This document defines a Historic Document for the Internet community. This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Not all documents approved by the IESG are a candidate for any level of Internet Standard; see Section 2 of RFC 5741.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc6101.

Copyright Notice

Table of Contents

1. Introduction

The primary goal of the SSL protocol is to provide privacy and reliability between two communicating applications. The protocol is composed of two layers. At the lowest level, layered on top of some reliable transport protocol (e.g., TCP [RFC0793]), is the SSL record protocol. The SSL record protocol is used for encapsulation of various higher level protocols. One such encapsulated protocol, the SSL handshake protocol, allows the server and client to authenticate each other and to negotiate an encryption algorithm and cryptographic keys before the application protocol transmits or receives its first byte of data. One advantage of SSL is that it is application protocol independent. A higher level protocol can layer on top of the SSL protocol transparently. The SSL protocol provides connection security that has three basic properties:

- The connection is private. Encryption is used after an initial handshake to define a secret key. Symmetric cryptography is used for data encryption (e.g., DES [DES], 3DES [3DES], RC4 [SCH]).

- The peer's identity can be authenticated using asymmetric, or public key, cryptography (e.g., RSA [RSA], DSS [DSS]).

- The connection is reliable. Message transport includes a message integrity check using a keyed Message Authentication Code (MAC) [RFC2104]. Secure hash functions (e.g., SHA, MD5) are used for MAC computations.

2. Goals

The goals of SSL protocol version 3.0, in order of their priority, are:

1. Cryptographic security

   SSL should be used to establish a secure connection between two parties.

2. Interoperability

   Independent programmers should be able to develop applications utilizing SSL 3.0 that will then be able to successfully exchange cryptographic parameters without knowledge of one another's code.

Note: It is not the case that all instances of SSL (even in the same application domain) will be able to successfully connect. For instance, if the server supports a particular hardware token, and the client does not have access to such a token, then the connection will not succeed.

3. Extensibility

SSL seeks to provide a framework into which new public key and bulk encryption methods can be incorporated as necessary. This will also accomplish two sub-goals: to prevent the need to create a new protocol (and risking the introduction of possible new weaknesses) and to avoid the need to implement an entire new security library.

4. Relative efficiency

Cryptographic operations tend to be highly CPU intensive, particularly public key operations. For this reason, the SSL protocol has incorporated an optional session caching scheme to reduce the number of connections that need to be established from scratch. Additionally, care has been taken to reduce network activity.

3. Goals of This Document

The SSL protocol version 3.0 specification is intended primarily for readers who will be implementing the protocol and those doing cryptographic analysis of it. The spec has been written with this in mind, and it is intended to reflect the needs of those two groups. For that reason, many of the algorithm-dependent data structures and rules are included in the body of the text (as opposed to in an appendix), providing easier access to them.

This document is not intended to supply any details of service definition or interface definition, although it does cover select areas of policy as they are required for the maintenance of solid security.

4. Presentation Language

This document deals with the formatting of data in an external representation. The following very basic and somewhat casually defined presentation syntax will be used. The syntax draws from several sources in its structure. Although it resembles the programming language "C" in its syntax and External Data Representation (XDR) [RFC1832] in both its syntax and intent, it

would be risky to draw too many parallels. The purpose of this presentation language is to document SSL only, not to have general application beyond that particular goal.

4.1. Basic Block Size

The representation of all data items is explicitly specified. The basic data block size is one byte (i.e., 8 bits). Multiple byte data items are concatenations of bytes, from left to right, from top to bottom. From the byte stream, a multi-byte item (a numeric in the example) is formed (using C notation) by:

```
value = (byte[0] << 8*(n-1)) | (byte[1] << 8*(n-2)) | ...
| byte[n-1];
```

This byte ordering for multi-byte values is the commonplace network byte order or big-endian format.

4.2. Miscellaneous

Comments begin with "/*" and end with "*/". Optional components are denoted by enclosing them in "[[ ]]" double brackets. Single-byte entities containing uninterpreted data are of type opaque.

4.3. Vectors

A vector (single dimensioned array) is a stream of homogeneous data elements. The size of the vector may be specified at documentation time or left unspecified until runtime. In either case, the length declares the number of bytes, not the number of elements, in the vector. The syntax for specifying a new type T' that is a fixed-length vector of type T is

```
T T'[n];
```

Here, T' occupies n bytes in the data stream, where n is a multiple of the size of T. The length of the vector is not included in the encoded stream.

In the following example, Datum is defined to be three consecutive bytes that the protocol does not interpret, while Data is three consecutive Datum, consuming a total of nine bytes.

```
opaque Datum[3];      /* three uninterpreted bytes */
Datum Data[9];        /* 3 consecutive 3 byte vectors */
```

Variable-length vectors are defined by specifying a subrange of legal lengths, inclusively, using the notation <floor..ceiling>. When encoded, the actual length precedes the vector's contents in the byte stream. The length will be in the form of a number consuming as many bytes as required to hold the vector's specified maximum (ceiling) length. A variable-length vector with an actual length field of zero is referred to as an empty vector.

```
T T'<floor..ceiling>;
```

In the following example, mandatory is a vector that must contain between 300 and 400 bytes of type opaque. It can never be empty. The actual length field consumes two bytes, a uint16, sufficient to represent the value 400 (see Section 4.4). On the other hand, longer can represent up to 800 bytes of data, or 400 uint16 elements, and it may be empty. Its encoding will include a two-byte actual length field prepended to the vector.

```
opaque mandatory<300..400>;
      /* length field is 2 bytes, cannot be empty */
uint16 longer<0..800>;
      /* zero to 400 16-bit unsigned integers */
```

4.4. Numbers

The basic numeric data type is an unsigned byte (uint8). All larger numeric data types are formed from fixed-length series of bytes concatenated as described in Section 4.1 and are also unsigned. The following numeric types are predefined.

```
uint8 uint16[2];
uint8 uint24[3];
uint8 uint32[4];
uint8 uint64[8];
```

4.5. Enumerateds

An additional sparse data type is available called enum. A field of type enum can only assume the values declared in the definition. Each definition is a different type. Only enumerateds of the same type may be assigned or compared. Every element of an enumerated must be assigned a value, as demonstrated in the following example. Since the elements of the enumerated are not ordered, they can be assigned any unique value, in any order.

```
enum { e1(v1), e2(v2), ... , en(vn), [[(n)]] } Te;
```

Enumerateds occupy as much space in the byte stream as would its maximal defined ordinal value. The following definition would cause one byte to be used to carry fields of type Color.

```
enum { red(3), blue(5), white(7) } Color;
```

Optionally, one may specify a value without its associated tag to force the width definition without defining a superfluous element. In the following example, Taste will consume two bytes in the data stream but can only assume the values 1, 2, or 4.

```
enum { sweet(1), sour(2), bitter(4), (32000) } Taste;
```

The names of the elements of an enumeration are scoped within the defined type. In the first example, a fully qualified reference to the second element of the enumeration would be Color.blue. Such qualification is not required if the target of the assignment is well specified.

```
Color color = Color.blue;     /* overspecified, legal */
Color color = blue;           /* correct, type implicit */
```

For enumerateds that are never converted to external representation, the numerical information may be omitted.

```
enum { low, medium, high } Amount;
```

4.6. Constructed Types

Structure types may be constructed from primitive types for convenience. Each specification declares a new, unique type. The syntax for definition is much like that of C.

```
struct {
    T1 f1;
    T2 f2;
    ...
    Tn fn;
} [[T]];
```

The fields within a structure may be qualified using the type's name using a syntax much like that available for enumerateds. For example, T.f2 refers to the second field of the previous declaration. Structure definitions may be embedded.

4.6.1. Variants

Defined structures may have variants based on some knowledge that is available within the environment. The selector must be an enumerated type that defines the possible variants the structure defines. There must be a case arm for every element of the enumeration declared in the select. The body of the variant structure may be given a label for reference. The mechanism by which the variant is selected at runtime is not prescribed by the presentation language.

```
struct {
    T1 f1;
    T2 f2;
     ....
    Tn fn;
    select (E) {
        case e1: Te1;
        case e2: Te2;
            ....
        case en: Ten;
    } [[fv]];
} [[Tv]];
```

For example,

```
enum { apple, orange } VariantTag;
struct {
    uint16 number;
    opaque string<0..10>; /* variable length */
} V1;

struct {
    uint32 number;
    opaque string[10];    /* fixed length */
} V2;
struct {
    select (VariantTag) { /* value of selector is implicit */
        case apple: V1;   /* VariantBody, tag = apple */
        case orange: V2;  /* VariantBody, tag = orange */
    } variant_body;       /* optional label on variant */
} VariantRecord;
```

Variant structures may be qualified (narrowed) by specifying a value for the selector prior to the type. For example, an

```
orange VariantRecord
```

is a narrowed type of a VariantRecord containing a variant_body of type V2.

4.7. Cryptographic Attributes

The four cryptographic operations digital signing, stream cipher encryption, block cipher encryption, and public key encryption are designated digitally-signed, stream-ciphered, block-ciphered, and public-key-encrypted, respectively. A field's cryptographic processing is specified by prepending an appropriate key word designation before the field's type specification. Cryptographic keys are implied by the current session state (see Section 5.1).

In digital signing, one-way hash functions are used as input for a signing algorithm. In RSA signing, a 36-byte structure of two hashes (one SHA and one MD5) is signed (encrypted with the private key). In DSS, the 20 bytes of the SHA hash are run directly through the Digital Signature Algorithm with no additional hashing.

In stream cipher encryption, the plaintext is exclusive-ORed with an identical amount of output generated from a cryptographically secure keyed pseudorandom number generator.

In block cipher encryption, every block of plaintext encrypts to a block of ciphertext. Because it is unlikely that the plaintext (whatever data is to be sent) will break neatly into the necessary block size (usually 64 bits), it is necessary to pad out the end of short blocks with some regular pattern, usually all zeroes.

In public key encryption, one-way functions with secret "trapdoors" are used to encrypt the outgoing data. Data encrypted with the public key of a given key pair can only be decrypted with the private key, and vice versa. In the following example:

```
stream-ciphered struct {
    uint8 field1;
    uint8 field2;
    digitally-signed opaque hash[20];
} UserType;
```

The contents of hash are used as input for the signing algorithm, then the entire structure is encrypted with a stream cipher.

4.8. Constants

Typed constants can be defined for purposes of specification by declaring a symbol of the desired type and assigning values to it. Under-specified types (opaque, variable-length vectors, and structures that contain opaque) cannot be assigned values. No fields of a multi-element structure or vector may be elided.

For example,

```
struct {
    uint8 f1;
    uint8 f2;
} Example1;

Example1 ex1 = {1, 4};/* assigns f1 = 1, f2 = 4 */
```

5. SSL Protocol

SSL is a layered protocol. At each layer, messages may include fields for length, description, and content. SSL takes messages to be transmitted, fragments the data into manageable blocks, optionally compresses the data, applies a MAC, encrypts, and transmits the result. Received data is decrypted, verified, decompressed, and reassembled, then delivered to higher level clients.

5.1. Session and Connection States

An SSL session is stateful. It is the responsibility of the SSL handshake protocol to coordinate the states of the client and server, thereby allowing the protocol state machines of each to operate consistently, despite the fact that the state is not exactly parallel. Logically, the state is represented twice, once as the current operating state and (during the handshake protocol) again as the pending state. Additionally, separate read and write states are maintained. When the client or server receives a change cipher spec message, it copies the pending read state into the current read state. When the client or server sends a change cipher spec message, it copies the pending write state into the current write state. When the handshake negotiation is complete, the client and server exchange change cipher spec messages (see Section 5.3), and they then communicate using the newly agreed-upon cipher spec.

An SSL session may include multiple secure connections; in addition, parties may have multiple simultaneous sessions.

The session state includes the following elements:

session identifier: An arbitrary byte sequence chosen by the server to identify an active or resumable session state.

peer certificate: X509.v3 [X509] certificate of the peer. This element of the state may be null.

compression method: The algorithm used to compress data prior to encryption.

cipher spec: Specifies the bulk data encryption algorithm (such as null, DES, etc.) and a MAC algorithm (such as MD5 or SHA). It also defines cryptographic attributes such as the hash_size. (See Appendix A.7 for formal definition.)

master secret: 48-byte secret shared between the client and server.

is resumable: A flag indicating whether the session can be used to initiate new connections.

The connection state includes the following elements:

server and client random: Byte sequences that are chosen by the server and client for each connection.

server write MAC secret: The secret used in MAC operations on data written by the server.

client write MAC secret: The secret used in MAC operations on data written by the client.

server write key: The bulk cipher key for data encrypted by the server and decrypted by the client.

client write key: The bulk cipher key for data encrypted by the client and decrypted by the server.

initialization vectors: When a block cipher in Cipher Block Chaining (CBC) mode is used, an initialization vector (IV) is maintained for each key. This field is first initialized by the SSL handshake protocol. Thereafter, the final ciphertext block from each record is preserved for use with the following record.

sequence numbers: Each party maintains separate sequence numbers for transmitted and received messages for each connection. When a party sends or receives a change cipher spec message, the appropriate sequence number is set to zero. Sequence numbers are of type uint64 and may not exceed 2^64-1.

5.2. Record Layer

The SSL record layer receives uninterpreted data from higher layers in non-empty blocks of arbitrary size.

5.2.1. Fragmentation

The record layer fragments information blocks into SSLPlaintext records of 2^14 bytes or less. Client message boundaries are not preserved in the record layer (i.e., multiple client messages of the same ContentType may be coalesced into a single SSLPlaintext record).

```
struct {
    uint8 major, minor;
} ProtocolVersion;

enum {
    change_cipher_spec(20), alert(21), handshake(22),
    application_data(23), (255)
} ContentType;

struct {
    ContentType type;
    ProtocolVersion version;
    uint16 length;
    opaque fragment[SSLPlaintext.length];
} SSLPlaintext;
```

type: The higher level protocol used to process the enclosed fragment.

version: The version of protocol being employed. This document describes SSL version 3.0 (see Appendix A.1).

length: The length (in bytes) of the following SSLPlaintext.fragment. The length should not exceed 2^14.

fragment: The application data. This data is transparent and treated as an independent block to be dealt with by the higher level protocol specified by the type field.

Note: Data of different SSL record layer content types may be interleaved. Application data is generally of lower precedence for transmission than other content types.

5.2.2. Record Compression and Decompression

All records are compressed using the compression algorithm defined in the current session state. There is always an active compression algorithm; however, initially it is defined as CompressionMethod.null. The compression algorithm translates an SSLPlaintext structure into an SSLCompressed structure. Compression functions erase their state information whenever the CipherSpec is replaced.

Note: The CipherSpec is part of the session state described in Section 5.1. References to fields of the CipherSpec are made throughout this document using presentation syntax. A more complete description of the CipherSpec is shown in Appendix A.7.

Compression must be lossless and may not increase the content length by more than 1024 bytes. If the decompression function encounters an SSLCompressed.fragment that would decompress to a length in excess of 2^14 bytes, it should issue a fatal decompression_failure alert (Section 5.4.2).

```
struct {
    ContentType type;       /* same as SSLPlaintext.type */
    ProtocolVersion version;/* same as SSLPlaintext.version */
    uint16 length;
    opaque fragment[SSLCompressed.length];
} SSLCompressed;
```

length: The length (in bytes) of the following SSLCompressed.fragment. The length should not exceed 2^14 + 1024.

fragment: The compressed form of SSLPlaintext.fragment.

Note: A CompressionMethod.null operation is an identity operation; no fields are altered (see Appendix A.4.1.)

Implementation note: Decompression functions are responsible for ensuring that messages cannot cause internal buffer overflows.

5.2.3. Record Payload Protection and the CipherSpec

All records are protected using the encryption and MAC algorithms defined in the current CipherSpec. There is always an active CipherSpec; however, initially it is SSL_NULL_WITH_NULL_NULL, which does not provide any security.

Once the handshake is complete, the two parties have shared secrets that are used to encrypt records and compute keyed Message Authentication Codes (MACs) on their contents. The techniques used to perform the encryption and MAC operations are defined by the CipherSpec and constrained by CipherSpec.cipher_type. The encryption and MAC functions translate an SSLCompressed structure into an SSLCiphertext. The decryption functions reverse the process. Transmissions also include a sequence number so that missing, altered, or extra messages are detectable.

```
struct {
    ContentType type;
    ProtocolVersion version;
    uint16 length;
    select (CipherSpec.cipher_type) {
        case stream: GenericStreamCipher;
        case block: GenericBlockCipher;
    } fragment;
} SSLCiphertext;
```

type: The type field is identical to SSLCompressed.type.

version: The version field is identical to SSLCompressed.version.

length: The length (in bytes) of the following SSLCiphertext.fragment. The length may not exceed 2^14 + 2048.

fragment: The encrypted form of SSLCompressed.fragment, including the MAC.

5.2.3.1. Null or Standard Stream Cipher

Stream ciphers (including BulkCipherAlgorithm.null; see Appendix A.7) convert SSLCompressed.fragment structures to and from stream SSLCiphertext.fragment structures.

```
stream-ciphered struct {
    opaque content[SSLCompressed.length];
    opaque MAC[CipherSpec.hash_size];
} GenericStreamCipher;
```

The MAC is generated as:

```
hash(MAC_write_secret + pad_2 +
     hash(MAC_write_secret + pad_1 + seq_num +
          SSLCompressed.type + SSLCompressed.length +
          SSLCompressed.fragment));
```

where "+" denotes concatenation.

pad_1: The character 0x36 repeated 48 times for MD5 or 40 times for SHA.

pad_2: The character 0x5c repeated 48 times for MD5 or 40 times for SHA.

seq_num: The sequence number for this message.

hash: Hashing algorithm derived from the cipher suite.

Note that the MAC is computed before encryption. The stream cipher encrypts the entire block, including the MAC. For stream ciphers that do not use a synchronization vector (such as RC4), the stream cipher state from the end of one record is simply used on the subsequent packet. If the CipherSuite is SSL_NULL_WITH_NULL_NULL, encryption consists of the identity operation (i.e., the data is not encrypted and the MAC size is zero implying that no MAC is used). SSLCiphertext.length is SSLCompressed.length plus CipherSpec.hash_size.

5.2.3.2. CBC Block Cipher

For block ciphers (such as RC2 or DES), the encryption and MAC functions convert SSLCompressed.fragment structures to and from block SSLCiphertext.fragment structures.

```
block-ciphered struct {
    opaque content[SSLCompressed.length];
    opaque MAC[CipherSpec.hash_size];
    uint8 padding[GenericBlockCipher.padding_length];
    uint8 padding_length;
} GenericBlockCipher;
```

The MAC is generated as described in Section 5.2.3.1.

padding: Padding that is added to force the length of the plaintext to be a multiple of the block cipher’s block length.

padding_length: The length of the padding must be less than the cipher's block length and may be zero. The padding length should be such that the total size of the GenericBlockCipher structure is a multiple of the cipher's block length.

The encrypted data length (SSLCiphertext.length) is one more than the sum of SSLCompressed.length, CipherSpec.hash_size, and padding_length.

Note: With CBC, the initialization vector (IV) for the first record is provided by the handshake protocol. The IV for subsequent records is the last ciphertext block from the previous record.

5.3. Change Cipher Spec Protocol

The change cipher spec protocol exists to signal transitions in ciphering strategies. The protocol consists of a single message, which is encrypted and compressed under the current (not the pending) CipherSpec. The message consists of a single byte of value 1.

```
struct {
    enum { change_cipher_spec(1), (255) } type;
} ChangeCipherSpec;
```

The change cipher spec message is sent by both the client and server to notify the receiving party that subsequent records will be protected under the just-negotiated CipherSpec and keys. Reception of this message causes the receiver to copy the read pending state into the read current state. The client sends a change cipher spec message following handshake key exchange and certificate verify messages (if any), and the server sends one after successfully processing the key exchange message it received from the client. An unexpected change cipher spec message should generate an unexpected_message alert (Section 5.4.2). When resuming a previous session, the change cipher spec message is sent after the hello messages.

5.4. Alert Protocol

One of the content types supported by the SSL record layer is the alert type. Alert messages convey the severity of the message and a description of the alert. Alert messages with a level of fatal result in the immediate termination of the connection. In this case, other connections corresponding to the session may continue, but the session identifier must be invalidated, preventing the failed session from being used to establish new connections. Like other messages, alert messages are encrypted and compressed, as specified by the current connection state.

```
enum { warning(1), fatal(2), (255) } AlertLevel;

enum {
    close_notify(0),
    unexpected_message(10),
    bad_record_mac(20),
    decompression_failure(30),
    handshake_failure(40),
    no_certificate(41),
    bad_certificate(42),
    unsupported_certificate(43),
    certificate_revoked(44),
    certificate_expired(45),
    certificate_unknown(46),
    illegal_parameter (47)
    (255)
} AlertDescription;

struct {
    AlertLevel level;
    AlertDescription description;
} Alert;
```

5.4.1. Closure Alerts

The client and the server must share knowledge that the connection is ending in order to avoid a truncation attack. Either party may initiate the exchange of closing messages.

close_notify: This message notifies the recipient that the sender will not send any more messages on this connection. The session becomes unresumable if any connection is terminated without proper close_notify messages with level equal to warning.

Either party may initiate a close by sending a close_notify alert. Any data received after a closure alert is ignored.

Each party is required to send a close_notify alert before closing the write side of the connection. It is required that the other party respond with a close_notify alert of its own and close down the connection immediately, discarding any pending writes. It is not required for the initiator of the close to wait for the responding close_notify alert before closing the read side of the connection.

NB: It is assumed that closing a connection reliably delivers pending data before destroying the transport.

5.4.2. Error Alerts

Error handling in the SSL handshake protocol is very simple. When an error is detected, the detecting party sends a message to the other party. Upon transmission or receipt of a fatal alert message, both parties immediately close the connection. Servers and clients are required to forget any session identifiers, keys, and secrets associated with a failed connection. The following error alerts are defined:

unexpected_message: An inappropriate message was received. This alert is always fatal and should never be observed in communication between proper implementations.

bad_record_mac: This alert is returned if a record is received with an incorrect MAC. This message is always fatal.

decompression_failure: The decompression function received improper input (e.g., data that would expand to excessive length). This message is always fatal.

handshake_failure: Reception of a handshake_failure alert message indicates that the sender was unable to negotiate an acceptable set of security parameters given the options available. This is a fatal error.

no_certificate: A no_certificate alert message may be sent in response to a certification request if no appropriate certificate is available.

bad_certificate: A certificate was corrupt, contained signatures that did not verify correctly, etc.

unsupported_certificate: A certificate was of an unsupported type.

certificate_revoked: A certificate was revoked by its signer.

certificate_expired: A certificate has expired or is not currently valid.

certificate_unknown: Some other (unspecified) issue arose in processing the certificate, rendering it unacceptable.

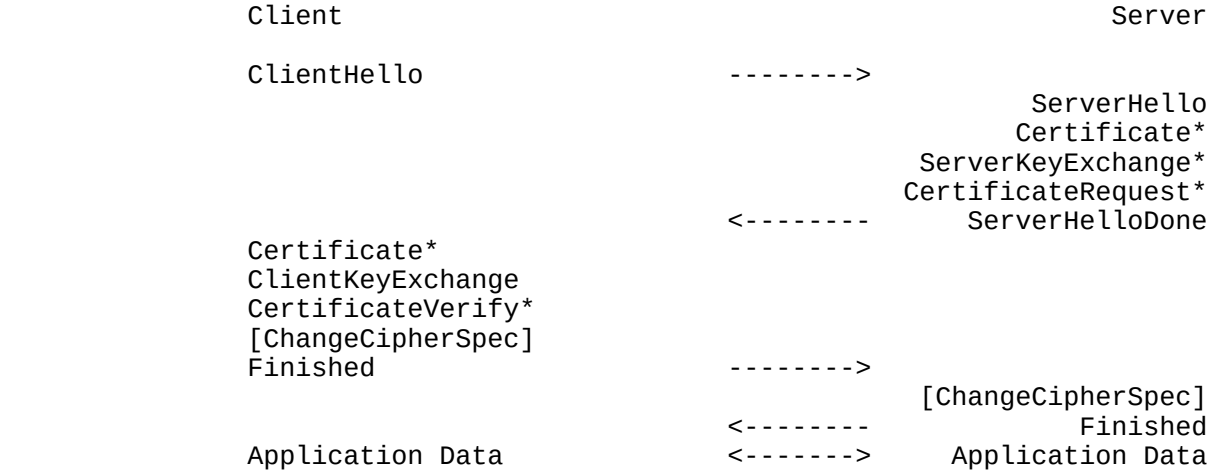
illegal_parameter: A field in the handshake was out of range or inconsistent with other fields. This is always fatal.

5.5. Handshake Protocol Overview

The cryptographic parameters of the session state are produced by the SSL handshake protocol, which operates on top of the SSL record layer. When an SSL client and server first start communicating, they agree on a protocol version, select cryptographic algorithms, optionally authenticate each other, and use public key encryption techniques to generate shared secrets. These processes are performed in the handshake protocol, which can be summarized as follows: the client sends a client hello message to which the server must respond with a server hello message, or else a fatal error will occur and the connection will fail. The client hello and server hello are used to establish security enhancement capabilities between client and server. The client hello and server hello establish the following attributes: Protocol Version, Session ID, Cipher Suite, and Compression Method. Additionally, two random values are generated and exchanged: ClientHello.random and ServerHello.random.

Following the hello messages, the server will send its certificate, if it is to be authenticated. Additionally, a server key exchange message may be sent, if it is required (e.g., if their server has no certificate, or if its certificate is for signing only). If the server is authenticated, it may request a certificate from the client, if that is appropriate to the cipher suite selected. Now the server will send the server hello done message, indicating that the hello-message phase of the handshake is complete. The server will then wait for a client response. If the server has sent a certificate request message, the client must send either the certificate message or a no_certificate alert. The client key exchange message is now sent, and the content of that message will depend on the public key algorithm selected between the client hello and the server hello. If the client has sent a certificate with signing ability, a digitally-signed certificate verify message is sent to explicitly verify the certificate.

At this point, a change cipher spec message is sent by the client, and the client copies the pending CipherSpec into the current CipherSpec. The client then immediately sends the finished message under the new algorithms, keys, and secrets. In response, the server will send its own change cipher spec message, transfer the pending to the current CipherSpec, and send its finished message under the new CipherSpec. At this point, the handshake is complete and the client and server may begin to exchange application layer data. (See flow chart below.)



* Indicates optional or situation-dependent messages that are not always sent.

Note: To help avoid pipeline stalls, ChangeCipherSpec is an independent SSL protocol content type, and is not actually an SSL handshake message.

When the client and server decide to resume a previous session or duplicate an existing session (instead of negotiating new security parameters) the message flow is as follows:

The client sends a ClientHello using the session ID of the session to be resumed. The server then checks its session cache for a match. If a match is found, and the server is willing to re-establish the connection under the specified session state, it will send a ServerHello with the same session ID value. At this point, both client and server must send change cipher spec messages and proceed directly to finished messages. Once the re-establishment is complete, the client and server may begin to exchange application layer data. (See flow chart below.) If a session ID match is not found, the server generates a new session ID and the SSL client and server perform a full handshake.

```
      Client                                               Server

      ClientHello                  -------->
                                                      ServerHello
                                                [change cipher spec]
                                   <--------             Finished
      change cipher spec
      Finished                     -------->
      Application Data             <------->     Application Data
```

The contents and significance of each message will be presented in detail in the following sections.

5.6. Handshake Protocol

The SSL handshake protocol is one of the defined higher level clients of the SSL record protocol. This protocol is used to negotiate the secure attributes of a session. Handshake messages are supplied to the SSL record layer, where they are encapsulated within one or more SSLPlaintext structures, which are processed and transmitted as specified by the current active session state.

```
enum {
    hello_request(0), client_hello(1), server_hello(2),
    certificate(11), server_key_exchange (12),
    certificate_request(13), server_hello_done(14),
    certificate_verify(15), client_key_exchange(16),
    finished(20), (255)
} HandshakeType;

struct {
    HandshakeType msg_type;    /* handshake type */
    uint24 length;             /* bytes in message */
    select (HandshakeType) {
        case hello_request: HelloRequest;
        case client_hello: ClientHello;
        case server_hello: ServerHello;
        case certificate: Certificate;
        case server_key_exchange: ServerKeyExchange;
        case certificate_request: CertificateRequest;
        case server_hello_done: ServerHelloDone;
        case certificate_verify: CertificateVerify;
        case client_key_exchange: ClientKeyExchange;
        case finished: Finished;
    } body;
} Handshake;
```

The handshake protocol messages are presented in the order they must be sent; sending handshake messages in an unexpected order results in a fatal error.

5.6.1. Hello messages

The hello phase messages are used to exchange security enhancement capabilities between the client and server. When a new session begins, the CipherSpec encryption, hash, and compression algorithms are initialized to null. The current CipherSpec is used for renegotiation messages.

5.6.1.1. Hello Request

The hello request message may be sent by the server at any time, but will be ignored by the client if the handshake protocol is already underway. It is a simple notification that the client should begin the negotiation process anew by sending a client hello message when convenient.

Note: Since handshake messages are intended to have transmission precedence over application data, it is expected that the negotiation begin in no more than one or two times the transmission time of a maximum-length application data message.

After sending a hello request, servers should not repeat the request until the subsequent handshake negotiation is complete. A client that receives a hello request while in a handshake negotiation state should simply ignore the message.

The structure of a hello request message is as follows:

```
struct { } HelloRequest;
```

5.6.1.2. Client Hello

When a client first connects to a server it is required to send the client hello as its first message. The client can also send a client hello in response to a hello request or on its own initiative in order to renegotiate the security parameters in an existing connection. The client hello message includes a random structure, which is used later in the protocol.

```
struct {
    uint32 gmt_unix_time;
    opaque random_bytes[28];
} Random;
```

gmt_unix_time: The current time and date in standard UNIX 32-bit format according to the sender's internal clock. Clocks are not required to be set correctly by the basic SSL protocol; higher level or application protocols may define additional requirements.

random_bytes: 28 bytes generated by a secure random number generator.

The client hello message includes a variable-length session identifier. If not empty, the value identifies a session between the same client and server whose security parameters the client wishes to reuse. The session identifier may be from an earlier connection, this connection, or another currently active connection. The second option is useful if the client only wishes to update the random structures and derived values of a connection, while the third option makes it possible to establish several simultaneous independent secure connections without repeating the full handshake protocol. The actual contents of the SessionID are defined by the server.

```
opaque SessionID<0..32>;
```

Warning: Servers must not place confidential information in session identifiers or let the contents of fake session identifiers cause any breach of security.

The CipherSuite list, passed from the client to the server in the client hello message, contains the combinations of cryptographic algorithms supported by the client in order of the client's preference (first choice first). Each CipherSuite defines both a key exchange algorithm and a CipherSpec. The server will select a cipher suite or, if no acceptable choices are presented, return a handshake failure alert and close the connection.

```
uint8 CipherSuite[2];  /* Cryptographic suite selector */
```

The client hello includes a list of compression algorithms supported by the client, ordered according to the client's preference. If the server supports none of those specified by the client, the session must fail.

```
enum { null(0), (255) } CompressionMethod;
```

Issue: Which compression methods to support is under investigation.

The structure of the client hello is as follows.

```
struct {
    ProtocolVersion client_version;
    Random random;
    SessionID session_id;
    CipherSuite cipher_suites<2..2^16-1>;
    CompressionMethod compression_methods<1..2^8-1>;
} ClientHello;
```

client_version: The version of the SSL protocol by which the client wishes to communicate during this session. This should be the most recent (highest valued) version supported by the client. For this version of the specification, the version will be 3.0 (see Appendix E for details about backward compatibility).

random: A client-generated random structure.

session_id: The ID of a session the client wishes to use for this connection. This field should be empty if no session_id is available or the client wishes to generate new security parameters.

cipher_suites: This is a list of the cryptographic options supported by the client, sorted with the client's first preference first. If the session_id field is not empty (implying a session resumption request), this vector must include at least the cipher_suite from that session. Values are defined in Appendix A.6.

compression_methods: This is a list of the compression methods supported by the client, sorted by client preference. If the session_id field is not empty (implying a session resumption request), this vector must include at least the compression_method from that session. All implementations must support CompressionMethod.null.

After sending the client hello message, the client waits for a server hello message. Any other handshake message returned by the server except for a hello request is treated as a fatal error.

Implementation note: Application data may not be sent before a finished message has been sent. Transmitted application data is known to be insecure until a valid finished message has been received. This absolute restriction is relaxed if there is a current, non-null encryption on this connection.

Forward compatibility note: In the interests of forward compatibility, it is permitted for a client hello message to include extra data after the compression methods. This data must be included in the handshake hashes, but must otherwise be ignored.

5.6.1.3. Server Hello

The server processes the client hello message and responds with either a handshake_failure alert or server hello message.

```
struct {
    ProtocolVersion server_version;
    Random random;
    SessionID session_id;
    CipherSuite cipher_suite;
    CompressionMethod compression_method;
} ServerHello;
```

server_version: This field will contain the lower of that suggested by the client in the client hello and the highest supported by the server. For this version of the specification, the version will be 3.0 (see Appendix E for details about backward compatibility).

random: This structure is generated by the server and must be different from (and independent of) ClientHello.random.

session_id: This is the identity of the session corresponding to this connection. If the ClientHello.session_id was non-empty, the server will look in its session cache for a match. If a match is found and the server is willing to establish the new connection using the specified session state, the server will respond with the same value as was supplied by the client. This indicates a resumed session and dictates that the parties must proceed directly to the finished messages. Otherwise, this field will contain a different value identifying the new session. The server may return an empty session_id to indicate that the session will not be cached and therefore cannot be resumed.

cipher_suite: The single cipher suite selected by the server from the list in ClientHello.cipher_suites. For resumed sessions, this field is the value from the state of the session being resumed.

compression_method: The single compression algorithm selected by the server from the list in ClientHello.compression_methods. For resumed sessions, this field is the value from the resumed session state.

5.6.2. Server Certificate

If the server is to be authenticated (which is generally the case), the server sends its certificate immediately following the server hello message. The certificate type must be appropriate for the selected cipher suite's key exchange algorithm, and is generally an X.509.v3 certificate (or a modified X.509 certificate in the case of FORTEZZA(tm) [FOR]). The same message type will be used for the client's response to a certificate request message.

```
opaque ASN.1Cert<1..2^24-1>;
struct {
    ASN.1Cert certificate_list<1..2^24-1>;
} Certificate;
```

certificate_list: This is a sequence (chain) of X.509.v3 certificates, ordered with the sender's certificate first followed by any certificate authority certificates proceeding sequentially upward.

Note: PKCS #7 [PKCS7] is not used as the format for the certificate vector because PKCS #6 [PKCS6] extended certificates are not used. Also, PKCS #7 defines a Set rather than a Sequence, making the task of parsing the list more difficult.

5.6.3. Server Key Exchange Message

The server key exchange message is sent by the server if it has no certificate, has a certificate only used for signing (e.g., DSS [DSS] certificates, signing-only RSA [RSA] certificates), or FORTEZZA KEA key exchange is used. This message is not used if the server certificate contains Diffie-Hellman [DH1] parameters.

Note: According to current US export law, RSA moduli larger than 512 bits may not be used for key exchange in software exported from the US. With this message, larger RSA keys may be used as signature-only certificates to sign temporary shorter RSA keys for key exchange.

```
enum { rsa, diffie_hellman, fortezza_kea }
       KeyExchangeAlgorithm;

struct {
    opaque rsa_modulus<1..2^16-1>;
    opaque rsa_exponent<1..2^16-1>;
} ServerRSAParams;
```

rsa_modulus: The modulus of the server's temporary RSA key.

rsa_exponent: The public exponent of the server's temporary RSA key.

```
struct {
    opaque dh_p<1..2^16-1>;
    opaque dh_g<1..2^16-1>;
    opaque dh_Ys<1..2^16-1>;
} ServerDHParams;     /* Ephemeral DH parameters */
```

dh_p: The prime modulus used for the Diffie-Hellman operation.

dh_g: The generator used for the Diffie-Hellman operation.

dh_Ys: The server's Diffie-Hellman public value (gX mod p).

```
struct {
    opaque r_s [128];
} ServerFortezzaParams;
```

r_s: Server random number for FORTEZZA KEA (Key Exchange Algorithm).

```
struct {
    select (KeyExchangeAlgorithm) {
        case diffie_hellman:
            ServerDHParams params;
            Signature signed_params;
        case rsa:
            ServerRSAParams params;
            Signature signed_params;
        case fortezza_kea:
            ServerFortezzaParams params;
    };
} ServerKeyExchange;
```

params: The server's key exchange parameters.

signed_params: A hash of the corresponding params value, with the signature appropriate to that hash applied.

md5_hash: MD5(ClientHello.random + ServerHello.random + ServerParams);

```
sha_hash:  SHA(ClientHello.random + ServerHello.random +
   ServerParams);

     enum { anonymous, rsa, dsa } SignatureAlgorithm;

     digitally-signed struct {
         select(SignatureAlgorithm) {
             case anonymous: struct { };
             case rsa:
                 opaque md5_hash[16];
                 opaque sha_hash[20];
             case dsa:
                 opaque sha_hash[20];
         };
     } Signature;
```

5.6.4. Certificate Request

A non-anonymous server can optionally request a certificate from the client, if appropriate for the selected cipher suite.

```
     enum {
         rsa_sign(1), dss_sign(2), rsa_fixed_dh(3), dss_fixed_dh(4),
         rsa_ephemeral_dh(5), dss_ephemeral_dh(6), fortezza_kea(20),
         (255)
     } ClientCertificateType;

     opaque DistinguishedName<1..2^16-1>;

     struct {
         ClientCertificateType certificate_types<1..2^8-1>;
         DistinguishedName certificate_authorities<3..2^16-1>;
     } CertificateRequest;
```

certificate_types: This field is a list of the types of certificates requested, sorted in order of the server's preference.

certificate_authorities: A list of the distinguished names of acceptable certificate authorities.

Note: DistinguishedName is derived from [X509].

Note: It is a fatal handshake_failure alert for an anonymous server to request client identification.

5.6.5. Server Hello Done

The server hello done message is sent by the server to indicate the end of the server hello and associated messages. After sending this message, the server will wait for a client response.

```
struct { } ServerHelloDone;
```

Upon receipt of the server hello done message the client should verify that the server provided a valid certificate if required and check that the server hello parameters are acceptable.

5.6.6. Client Certificate

This is the first message the client can send after receiving a server hello done message. This message is only sent if the server requests a certificate. If no suitable certificate is available, the client should send a no_certificate alert instead. This alert is only a warning; however, the server may respond with a fatal handshake failure alert if client authentication is required. Client certificates are sent using the certificate defined in Section 5.6.2.

Note: Client Diffie-Hellman certificates must match the server specified Diffie-Hellman parameters.

5.6.7. Client Key Exchange Message

The choice of messages depends on which public key algorithm(s) has (have) been selected. See Section 5.6.3 for the KeyExchangeAlgorithm definition.

```
struct {
    select (KeyExchangeAlgorithm) {
        case rsa: EncryptedPreMasterSecret;
        case diffie_hellman: ClientDiffieHellmanPublic;
        case fortezza_kea: FortezzaKeys;
    } exchange_keys;
} ClientKeyExchange;
```

The information to select the appropriate record structure is in the pending session state (see Section 5.1).

5.6.7.1. RSA Encrypted Premaster Secret Message

If RSA is being used for key agreement and authentication, the client generates a 48-byte premaster secret, encrypts it under the public key from the server's certificate or temporary RSA key from a server key exchange message, and sends the result in an encrypted premaster secret message.

```
struct {
    ProtocolVersion client_version;
    opaque random[46];
} PreMasterSecret;
```

client_version: The latest (newest) version supported by the client. This is used to detect version roll-back attacks.

random: 46 securely-generated random bytes.

```
struct {
    public-key-encrypted PreMasterSecret pre_master_secret;
} EncryptedPreMasterSecret;
```

pre_master_secret: This random value is generated by the client and is used to generate the master secret, as specified in Section 6.1.

5.6.7.2. FORTEZZA Key Exchange Message

Under FORTEZZA, the client derives a token encryption key (TEK) using the FORTEZZA Key Exchange Algorithm (KEA). The client's KEA calculation uses the public key in the server's certificate along with private parameters in the client's token. The client sends public parameters needed for the server to generate the TEK, using its own private parameters. The client generates session keys, wraps them using the TEK, and sends the results to the server. The client generates IVs for the session keys and TEK and sends them also. The client generates a random 48-byte premaster secret, encrypts it using the TEK, and sends the result:

```
struct {
    opaque y_c<0..128>;
    opaque r_c[128];
    opaque y_signature[40];
    opaque wrapped_client_write_key[12];
    opaque wrapped_server_write_key[12];
    opaque client_write_iv[24];
    opaque server_write_iv[24];
    opaque master_secret_iv[24];
    block-ciphered opaque encrypted_pre_master_secret[48];
} FortezzaKeys;
```

y_signature: y_signature is the signature of the KEA public key, signed with the client's DSS private key.

y_c: The client's Yc value (public key) for the KEA calculation. If the client has sent a certificate, and its KEA public key is suitable, this value must be empty since the certificate already contains this value. If the client sent a certificate without a suitable public key, y_c is used and y_signature is the KEA public key signed with the client's DSS private key. For this value to be used, it must be between 64 and 128 bytes.

r_c: The client's Rc value for the KEA calculation.

wrapped_client_write_key: This is the client's write key, wrapped by the TEK.

wrapped_server_write_key: This is the server's write key, wrapped by the TEK.

client_write_iv: The IV for the client write key.

server_write_iv: The IV for the server write key.

master_secret_iv: This is the IV for the TEK used to encrypt the premaster secret.

pre_master_secret: A random value, generated by the client and used to generate the master secret, as specified in Section 6.1. In the above structure, it is encrypted using the TEK.

5.6.7.3. Client Diffie-Hellman Public Value

This structure conveys the client's Diffie-Hellman public value (Yc) if it was not already included in the client's certificate. The encoding used for Yc is determined by the enumerated PublicValueEncoding.

```
enum { implicit, explicit } PublicValueEncoding;
```

implicit: If the client certificate already contains the public value, then it is implicit and Yc does not need to be sent again.

explicit: Yc needs to be sent.

```
struct {
    select (PublicValueEncoding) {
        case implicit: struct { };
        case explicit: opaque dh_Yc<1..2^16-1>;
    } dh_public;
} ClientDiffieHellmanPublic;
```

dh_Yc: The client's Diffie-Hellman public value (Yc).

5.6.8. Certificate Verify

This message is used to provide explicit verification of a client certificate. This message is only sent following any client certificate that has signing capability (i.e., all certificates except those containing fixed Diffie-Hellman parameters).

```
struct {
     Signature signature;
} CertificateVerify;
```

```
CertificateVerify.signature.md5_hash
           MD5(master_secret + pad_2 +
               MD5(handshake_messages + master_secret + pad_1));
Certificate.signature.sha_hash
           SHA(master_secret + pad_2 +
               SHA(handshake_messages + master_secret + pad_1));
```

pad_1: This is identical to the pad_1 defined in Section 5.2.3.1.

pad_2: This is identical to the pad_2 defined in Section 5.2.3.1.

Here, handshake_messages refers to all handshake messages starting at client hello up to but not including this message.

5.6.9. Finished

A finished message is always sent immediately after a change cipher spec message to verify that the key exchange and authentication processes were successful. The finished message is the first protected with the just-negotiated algorithms, keys, and secrets. No acknowledgment of the finished message is required; parties may begin sending encrypted data immediately after sending the finished message. Recipients of finished messages must verify that the contents are correct.

```
enum { client(0x434C4E54), server(0x53525652) } Sender;

struct {
    opaque md5_hash[16];
    opaque sha_hash[20];
} Finished;
```

md5_hash: MD5(master_secret + pad2 + MD5(handshake_messages + Sender + master_secret + pad1));

sha_hash: SHA(master_secret + pad2 + SHA(handshake_messages + Sender + master_secret + pad1));

handshake_messages: All of the data from all handshake messages up to but not including this message. This is only data visible at the handshake layer and does not include record layer headers.

It is a fatal error if a finished message is not preceeded by a change cipher spec message at the appropriate point in the handshake.

The hash contained in finished messages sent by the server incorporate Sender.server; those sent by the client incorporate Sender.client. The value handshake_messages includes all handshake messages starting at client hello up to but not including this finished message. This may be different from handshake_messages in Section 5.6.8 because it would include the certificate verify message (if sent).

Note: Change cipher spec messages are not handshake messages and are not included in the hash computations.

5.7. Application Data Protocol

Application data messages are carried by the record layer and are fragmented, compressed, and encrypted based on the current connection state. The messages are treated as transparent data to the record layer.

6. Cryptographic Computations

The key exchange, authentication, encryption, and MAC algorithms are determined by the cipher_suite selected by the server and revealed in the server hello message.

6.1. Asymmetric Cryptographic Computations

The asymmetric algorithms are used in the handshake protocol to authenticate parties and to generate shared keys and secrets.

For Diffie-Hellman, RSA, and FORTEZZA, the same algorithm is used to convert the pre_master_secret into the master_secret. The pre_master_secret should be deleted from memory once the master_secret has been computed.

```
master_secret =
  MD5(pre_master_secret + SHA('A' + pre_master_secret +
      ClientHello.random + ServerHello.random)) +
  MD5(pre_master_secret + SHA('BB' + pre_master_secret +
      ClientHello.random + ServerHello.random)) +
  MD5(pre_master_secret + SHA('CCC' + pre_master_secret +
      ClientHello.random + ServerHello.random));
```

6.1.1. RSA

When RSA is used for server authentication and key exchange, a 48-byte pre_master_secret is generated by the client, encrypted under the server's public key, and sent to the server. The server uses its private key to decrypt the pre_master_secret. Both parties then convert the pre_master_secret into the master_secret, as specified above.

RSA digital signatures are performed using PKCS #1 [PKCS1] block type 1. RSA public key encryption is performed using PKCS #1 block type 2.

6.1.2. Diffie-Hellman

A conventional Diffie-Hellman computation is performed. The negotiated key (Z) is used as the pre_master_secret, and is converted into the master_secret, as specified above.

Note: Diffie-Hellman parameters are specified by the server, and may be either ephemeral or contained within the server's certificate.

6.1.3. FORTEZZA

A random 48-byte pre_master_secret is sent encrypted under the TEK and its IV. The server decrypts the pre_master_secret and converts it into a master_secret, as specified above. Bulk cipher keys and IVs for encryption are generated by the client's token and exchanged in the key exchange message; the master_secret is only used for MAC computations.

6.2. Symmetric Cryptographic Calculations and the CipherSpec

The technique used to encrypt and verify the integrity of SSL records is specified by the currently active CipherSpec. A typical example would be to encrypt data using DES and generate authentication codes using MD5. The encryption and MAC algorithms are set to SSL_NULL_WITH_NULL_NULL at the beginning of the SSL handshake protocol, indicating that no message authentication or encryption is performed. The handshake protocol is used to negotiate a more secure CipherSpec and to generate cryptographic keys.

6.2.1. The Master Secret

Before secure encryption or integrity verification can be performed on records, the client and server need to generate shared secret information known only to themselves. This value is a 48-byte quantity called the master secret. The master secret is used to generate keys and secrets for encryption and MAC computations. Some algorithms, such as FORTEZZA, may have their own procedure for generating encryption keys (the master secret is used only for MAC computations in FORTEZZA).

6.2.2. Converting the Master Secret into Keys and MAC Secrets

The master secret is hashed into a sequence of secure bytes, which are assigned to the MAC secrets, keys, and non-export IVs required by the current CipherSpec (see Appendix A.7). CipherSpecs require a client write MAC secret, a server write MAC secret, a client write key, a server write key, a client write IV, and a server write IV, which are generated from the master secret in that order. Unused

values, such as FORTEZZA keys communicated in the KeyExchange message, are empty. The following inputs are available to the key definition process:

```
opaque MasterSecret[48]
ClientHello.random
ServerHello.random
```

When generating keys and MAC secrets, the master secret is used as an entropy source, and the random values provide unencrypted salt material and IVs for exportable ciphers.

To generate the key material, compute

```
key_block =
  MD5(master_secret + SHA('A' + master_secret +
                          ServerHello.random +
                          ClientHello.random)) +
  MD5(master_secret + SHA('BB' + master_secret +
                          ServerHello.random +
                          ClientHello.random)) +
  MD5(master_secret + SHA('CCC' + master_secret +
                          ServerHello.random +
                          ClientHello.random)) + [...];
```

until enough output has been generated. Then, the key_block is partitioned as follows.

```
client_write_MAC_secret[CipherSpec.hash_size]
server_write_MAC_secret[CipherSpec.hash_size]
client_write_key[CipherSpec.key_material]
server_write_key[CipherSpec.key_material]
client_write_IV[CipherSpec.IV_size] /* non-export ciphers */
server_write_IV[CipherSpec.IV_size] /* non-export ciphers */
```

Any extra key_block material is discarded.

Exportable encryption algorithms (for which CipherSpec.is_exportable is true) require additional processing as follows to derive their final write keys:

```
final_client_write_key = MD5(client_write_key +
                             ClientHello.random +
                             ServerHello.random);
final_server_write_key = MD5(server_write_key +
                             ServerHello.random +
                             ClientHello.random);
```

Exportable encryption algorithms derive their IVs from the random messages:

```
client_write_IV = MD5(ClientHello.random + ServerHello.random);
server_write_IV = MD5(ServerHello.random + ClientHello.random);
```

MD5 outputs are trimmed to the appropriate size by discarding the least-significant bytes.

6.2.2.1. Export Key Generation Example

SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5 requires five random bytes for each of the two encryption keys and 16 bytes for each of the MAC keys, for a total of 42 bytes of key material. MD5 produces 16 bytes of output per call, so three calls to MD5 are required. The MD5 outputs are concatenated into a 48-byte key_block with the first MD5 call providing bytes zero through 15, the second providing bytes 16 through 31, etc. The key_block is partitioned, and the write keys are salted because this is an exportable encryption algorithm.

```
client_write_MAC_secret = key_block[0..15]
server_write_MAC_secret = key_block[16..31]
client_write_key      = key_block[32..36]
server_write_key      = key_block[37..41]
final_client_write_key = MD5(client_write_key +
                             ClientHello.random +
                             ServerHello.random)[0..15];
final_server_write_key = MD5(server_write_key +
                             ServerHello.random +
                             ClientHello.random)[0..15];
client_write_IV = MD5(ClientHello.random +
                      ServerHello.random)[0..7];
server_write_IV = MD5(ServerHello.random +
                      ClientHello.random)[0..7];
```

7. Security Considerations

See Appendix F.

8. Informative References

[DH1] Diffie, W. and M. Hellman, "New Directions in Cryptography", IEEE Transactions on Information Theory V. IT-22, n. 6, pp. 74-84, June 1977.

[SSL-2] Hickman, K., "The SSL Protocol", February 1995.

[3DES] Tuchman, W., "Hellman Presents No Shortcut Solutions To DES", IEEE Spectrum, v. 16, n. 7, pp 40-41, July 1979.

[DES] ANSI X3.106, "American National Standard for Information Systems-Data Link Encryption", American National Standards Institute, 1983.

[DSS] NIST FIPS PUB 186, "Digital Signature Standard", National Institute of Standards and Technology U.S. Department of Commerce, May 1994.

[FOR] NSA X22, "FORTEZZA: Application Implementers Guide", Document # PD4002103-1.01, April 1995.

[RFC0959] Postel, J. and J. Reynolds, "File Transfer Protocol", STD 9, RFC 959, October 1985.

[RFC0791] Postel, J., "Internet Protocol", STD 5, RFC 791, September 1981.

[RFC1945] Berners-Lee, T., Fielding, R., and H. Nielsen, "Hypertext Transfer Protocol -- HTTP/1.0", RFC 1945, May 1996.

[RFC1321] Rivest, R., "The MD5 Message-Digest Algorithm", RFC 1321, April 1992.

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.

[RFC0854] Postel, J. and J. Reynolds, "Telnet Protocol Specification", STD 8, RFC 854, May 1983.

[RFC1832] Srinivasan, R., "XDR: External Data Representation Standard", RFC 1832, August 1995.

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, February 1997.

[IDEA] Lai, X., "On the Design and Security of Block Ciphers", ETH Series in Information Processing, v. 1, Konstanz: Hartung-Gorre Verlag, 1992.

[PKCS1] RSA Laboratories, "PKCS #1: RSA Encryption Standard version 1.5", November 1993.

[PKCS6] RSA Laboratories, "PKCS #6: RSA Extended Certificate Syntax Standard version 1.5", November 1993.

[PKCS7] RSA Laboratories, "PKCS #7: RSA Cryptographic Message Syntax Standard version 1.5", November 1993.

[RSA] Rivest, R., Shamir, A., and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", Communications of the ACM v. 21, n. 2 pp. 120-126., February 1978.

[SCH] Schneier, B., "Applied Cryptography: Protocols, Algorithms, and Source Code in C", John Wiley & Sons, 1994.

[SHA] NIST FIPS PUB 180-1, "Secure Hash Standard", May 1994.

National Institute of Standards and Technology, U.S. Department of Commerce, DRAFT

[X509] CCITT, "The Directory - Authentication Framework", Recommendation X.509 , 1988.

[RSADSI] RSA Data Security, Inc., "Unpublished works".

Appendix A. Protocol Constant Values

This section describes protocol types and constants.

A.1. Record Layer

```
struct {
    uint8 major, minor;
} ProtocolVersion;

ProtocolVersion version = { 3,0 };

enum {
    change_cipher_spec(20), alert(21), handshake(22),
    application_data(23), (255)
} ContentType;

struct {
    ContentType type;
    ProtocolVersion version;
    uint16 length;
    opaque fragment[SSLPlaintext.length];
} SSLPlaintext;

struct {
    ContentType type;
    ProtocolVersion version;
    uint16 length;
    opaque fragment[SSLCompressed.length];
} SSLCompressed;

struct {
    ContentType type;
    ProtocolVersion version;
    uint16 length;
    select (CipherSpec.cipher_type) {
        case stream: GenericStreamCipher;
        case block:  GenericBlockCipher;
    } fragment;
} SSLCiphertext;

stream-ciphered struct {
    opaque content[SSLCompressed.length];
    opaque MAC[CipherSpec.hash_size];
} GenericStreamCipher;

block-ciphered struct {
    opaque content[SSLCompressed.length];
```

```
            opaque MAC[CipherSpec.hash_size];
            uint8 padding[GenericBlockCipher.padding_length];
            uint8 padding_length;
        } GenericBlockCipher;
```

A.2. Change Cipher Specs Message

```
        struct {
            enum { change_cipher_spec(1), (255) } type;
        } ChangeCipherSpec;
```

A.3. Alert Messages

```
        enum { warning(1), fatal(2), (255) } AlertLevel;

        enum {
            close_notify(0),
            unexpected_message(10),
            bad_record_mac(20),
            decompression_failure(30),
            handshake_failure(40),
            no_certificate(41),
            bad_certificate(42),
            unsupported_certificate(43),
            certificate_revoked(44),
            certificate_expired(45),
            certificate_unknown(46),
            illegal_parameter (47),
            (255)
        } AlertDescription;

        struct {
            AlertLevel level;
            AlertDescription description;
        } Alert;
```

A.4. Handshake Protocol

```
      enum {
          hello_request(0), client_hello(1), server_hello(2),
          certificate(11), server_key_exchange (12),
          certificate_request(13), server_done(14),
          certificate_verify(15), client_key_exchange(16),
          finished(20), (255)
      } HandshakeType;

        struct {
            HandshakeType msg_type;
            uint24 length;
            select (HandshakeType) {
                case hello_request: HelloRequest;
                case client_hello: ClientHello;
                case server_hello: ServerHello;
                case certificate: Certificate;
                case server_key_exchange: ServerKeyExchange;
                case certificate_request: CertificateRequest;
                case server_done: ServerHelloDone;
                case certificate_verify: CertificateVerify;
                case client_key_exchange: ClientKeyExchange;
                case finished: Finished;
            } body;
        } Handshake;
```

A.4.1. Hello Messages

```
        struct { } HelloRequest;

        struct {
            uint32 gmt_unix_time;
            opaque random_bytes[28];
        } Random;

        opaque SessionID<0..32>;

        uint8 CipherSuite[2];

        enum { null(0), (255) } CompressionMethod;

        struct {
            ProtocolVersion client_version;
            Random random;
            SessionID session_id;
            CipherSuite cipher_suites<0..2^16-1>;
            CompressionMethod compression_methods<0..2^8-1>;
```

```
        } ClientHello;

        struct {
            ProtocolVersion server_version;
            Random random;
            SessionID session_id;
            CipherSuite cipher_suite;
            CompressionMethod compression_method;
        } ServerHello;
```

A.4.2. Server Authentication and Key Exchange Messages

```
        opaque ASN.1Cert<2^24-1>;

        struct {
            ASN.1Cert certificate_list<1..2^24-1>;
        } Certificate;

        enum { rsa, diffie_hellman, fortezza_kea } KeyExchangeAlgorithm;

        struct {
            opaque RSA_modulus<1..2^16-1>;
            opaque RSA_exponent<1..2^16-1>;
        } ServerRSAParams;

        struct {
            opaque DH_p<1..2^16-1>;
            opaque DH_g<1..2^16-1>;
            opaque DH_Ys<1..2^16-1>;
        } ServerDHParams;

        struct {
            opaque r_s [128]
        } ServerFortezzaParams

        struct {
            select (KeyExchangeAlgorithm) {
                case diffie_hellman:
                    ServerDHParams params;
                    Signature signed_params;
                case rsa:
                    ServerRSAParams params;
                    Signature signed_params;
                case fortezza_kea:
                    ServerFortezzaParams params;
            };
        } ServerKeyExchange;
```

```
        enum { anonymous, rsa, dsa } SignatureAlgorithm;

        digitally-signed struct {
            select(SignatureAlgorithm) {
                case anonymous: struct { };
                case rsa:
                    opaque md5_hash[16];
                    opaque sha_hash[20];
                case dsa:
                    opaque sha_hash[20];
            };
        } Signature;


        enum {
            RSA_sign(1), DSS_sign(2), RSA_fixed_DH(3),
            DSS_fixed_DH(4), RSA_ephemeral_DH(5), DSS_ephemeral_DH(6),
            FORTEZZA_MISSI(20), (255)
        } CertificateType;

        opaque DistinguishedName<1..2^16-1>;

        struct {
            CertificateType certificate_types<1..2^8-1>;
            DistinguishedName certificate_authorities<3..2^16-1>;
        } CertificateRequest;

        struct { } ServerHelloDone;
```

A.5. Client Authentication and Key Exchange Messages

```
        struct {
            select (KeyExchangeAlgorithm) {
                case rsa: EncryptedPreMasterSecret;
                case diffie_hellman: DiffieHellmanClientPublicValue;
                case fortezza_kea: FortezzaKeys;
            } exchange_keys;
        } ClientKeyExchange;

        struct {
            ProtocolVersion client_version;
            opaque random[46];
        } PreMasterSecret;

        struct {
            public-key-encrypted PreMasterSecret pre_master_secret;
        } EncryptedPreMasterSecret;
```

```
        struct {
            opaque y_c<0..128>;
            opaque r_c[128];
            opaque y_signature[40];
            opaque wrapped_client_write_key[12];
            opaque wrapped_server_write_key[12];
            opaque client_write_iv[24];
            opaque server_write_iv[24];
            opaque master_secret_iv[24];
            opaque encrypted_preMasterSecret[48];
        } FortezzaKeys;

        enum { implicit, explicit } PublicValueEncoding;

        struct {
            select (PublicValueEncoding) {
                case implicit: struct {};
                case explicit: opaque DH_Yc<1..2^16-1>;
            } dh_public;
        } ClientDiffieHellmanPublic;

        struct {
            Signature signature;
        } CertificateVerify;
```

A.5.1. Handshake Finalization Message

```
        struct {
            opaque md5_hash[16];
            opaque sha_hash[20];
        } Finished;
```

A.6. The CipherSuite

The following values define the CipherSuite codes used in the client hello and server hello messages.

A CipherSuite defines a cipher specifications supported in SSL version 3.0.

```
  CipherSuite SSL_NULL_WITH_NULL_NULL                = { 0x00,0x00 };
```

The following CipherSuite definitions require that the server provide an RSA certificate that can be used for key exchange. The server may request either an RSA or a DSS signature-capable certificate in the certificate request message.

```
  CipherSuite SSL_RSA_WITH_NULL_MD5                  = { 0x00,0x01 };
  CipherSuite SSL_RSA_WITH_NULL_SHA                  = { 0x00,0x02 };
  CipherSuite SSL_RSA_EXPORT_WITH_RC4_40_MD5         = { 0x00,0x03 };
  CipherSuite SSL_RSA_WITH_RC4_128_MD5               = { 0x00,0x04 };
  CipherSuite SSL_RSA_WITH_RC4_128_SHA               = { 0x00,0x05 };
  CipherSuite SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5     = { 0x00,0x06 };
  CipherSuite SSL_RSA_WITH_IDEA_CBC_SHA              = { 0x00,0x07 };
  CipherSuite SSL_RSA_EXPORT_WITH_DES40_CBC_SHA      = { 0x00,0x08 };
  CipherSuite SSL_RSA_WITH_DES_CBC_SHA               = { 0x00,0x09 };
  CipherSuite SSL_RSA_WITH_3DES_EDE_CBC_SHA          = { 0x00,0x0A };
```

The following CipherSuite definitions are used for server-authenticated (and optionally client-authenticated) Diffie-Hellman. DH denotes cipher suites in which the server's certificate contains the Diffie-Hellman parameters signed by the certificate authority (CA). DHE denotes ephemeral Diffie-Hellman, where the Diffie-Hellman parameters are signed by a DSS or RSA certificate, which has been signed by the CA. The signing algorithm used is specified after the DH or DHE parameter. In all cases, the client must have the same type of certificate, and must use the Diffie-Hellman parameters chosen by the server.

```
  CipherSuite SSL_DH_DSS_EXPORT_WITH_DES40_CBC_SHA   = { 0x00,0x0B };
  CipherSuite SSL_DH_DSS_WITH_DES_CBC_SHA            = { 0x00,0x0C };
  CipherSuite SSL_DH_DSS_WITH_3DES_EDE_CBC_SHA       = { 0x00,0x0D };
  CipherSuite SSL_DH_RSA_EXPORT_WITH_DES40_CBC_SHA   = { 0x00,0x0E };
  CipherSuite SSL_DH_RSA_WITH_DES_CBC_SHA            = { 0x00,0x0F };
  CipherSuite SSL_DH_RSA_WITH_3DES_EDE_CBC_SHA       = { 0x00,0x10 };
  CipherSuite SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA  = { 0x00,0x11 };
  CipherSuite SSL_DHE_DSS_WITH_DES_CBC_SHA           = { 0x00,0x12 };
  CipherSuite SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA      = { 0x00,0x13 };
  CipherSuite SSL_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA  = { 0x00,0x14 };
  CipherSuite SSL_DHE_RSA_WITH_DES_CBC_SHA           = { 0x00,0x15 };
  CipherSuite SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA      = { 0x00,0x16 };
```

The following cipher suites are used for completely anonymous Diffie-Hellman communications in which neither party is authenticated. Note that this mode is vulnerable to man-in-the-middle attacks and is therefore strongly discouraged.

```
  CipherSuite SSL_DH_anon_EXPORT_WITH_RC4_40_MD5     = { 0x00,0x17 };
  CipherSuite SSL_DH_anon_WITH_RC4_128_MD5           = { 0x00,0x18 };
  CipherSuite SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA  = { 0x00,0x19 };
  CipherSuite SSL_DH_anon_WITH_DES_CBC_SHA           = { 0x00,0x1A };
  CipherSuite SSL_DH_anon_WITH_3DES_EDE_CBC_SHA      = { 0x00,0x1B };
```

The final cipher suites are for the FORTEZZA token.

```
  CipherSuite SSL_FORTEZZA_KEA_WITH_NULL_SHA         = { 0X00,0X1C };
  CipherSuite SSL_FORTEZZA_KEA_WITH_FORTEZZA_CBC_SHA = { 0x00,0x1D };
  CipherSuite SSL_FORTEZZA_KEA_WITH_RC4_128_SHA      = { 0x00,0x1E };
```

Note: All cipher suites whose first byte is 0xFF are considered private and can be used for defining local/experimental algorithms. Interoperability of such types is a local matter.

A.7. The CipherSpec

A cipher suite identifies a CipherSpec. These structures are part of the SSL session state. The CipherSpec includes:

```
enum { stream, block } CipherType;

enum { true, false } IsExportable;

enum { null, rc4, rc2, des, 3des, des40, fortezza }
    BulkCipherAlgorithm;

enum { null, md5, sha } MACAlgorithm;

struct {
    BulkCipherAlgorithm bulk_cipher_algorithm;
    MACAlgorithm mac_algorithm;
    CipherType cipher_type;
    IsExportable is_exportable
    uint8 hash_size;
    uint8 key_material;
    uint8 IV_size;
} CipherSpec;
```

Appendix B. Glossary

application protocol: An application protocol is a protocol that normally layers directly on top of the transport layer (e.g., TCP/IP [RFC0793]/[RFC0791]). Examples include HTTP [RFC1945], TELNET [RFC0959], FTP [RFC0854], and SMTP.

asymmetric cipher: See public key cryptography.

authentication: Authentication is the ability of one entity to determine the identity of another entity.

block cipher: A block cipher is an algorithm that operates on plaintext in groups of bits, called blocks. 64 bits is a typical block size.

bulk cipher: A symmetric encryption algorithm used to encrypt large quantities of data.

cipher block chaining (CBC) mode: CBC is a mode in which every plaintext block encrypted with the block cipher is first exclusive-ORed with the previous ciphertext block (or, in the case of the first block, with the initialization vector).

certificate: As part of the X.509 protocol (a.k.a. ISO Authentication framework), certificates are assigned by a trusted certificate authority and provide verification of a party's identity and may also supply its public key.

client: The application entity that initiates a connection to a server.

client write key: The key used to encrypt data written by the client.

client write MAC secret: The secret data used to authenticate data written by the client.

connection: A connection is a transport (in the OSI layering model definition) that provides a suitable type of service. For SSL, such connections are peer-to-peer relationships. The connections are transient. Every connection is associated with one session.

Data Encryption Standard (DES): DES is a very widely used symmetric encryption algorithm. DES is a block cipher [DES] [3DES].

Digital Signature Standard: (DSS) A standard for digital signing, including the Digital Signature Algorithm, approved by the National Institute of Standards and Technology, defined in NIST FIPS PUB 186, "Digital Signature Standard," published May, 1994 by the U.S. Dept. of Commerce.

digital signatures: Digital signatures utilize public key cryptography and one-way hash functions to produce a signature of the data that can be authenticated, and is difficult to forge or repudiate.

FORTEZZA: A PCMCIA card that provides both encryption and digital signing.

handshake: An initial negotiation between client and server that establishes the parameters of their transactions.

Initialization Vector (IV): When a block cipher is used in CBC mode, the initialization vector is exclusive-ORed with the first plaintext block prior to encryption.

IDEA: A 64-bit block cipher designed by Xuejia Lai and James Massey [IDEA].

Message Authentication Code (MAC): A Message Authentication Code is a one-way hash computed from a message and some secret data. Its purpose is to detect if the message has been altered.

master secret: Secure secret data used for generating encryption keys, MAC secrets, and IVs.

MD5: MD5 [RFC1321] is a secure hashing function that converts an arbitrarily long data stream into a digest of fixed size.

public key cryptography: A class of cryptographic techniques employing two-key ciphers. Messages encrypted with the public key can only be decrypted with the associated private key. Conversely, messages signed with the private key can be verified with the public key.

one-way hash function: A one-way transformation that converts an arbitrary amount of data into a fixed-length hash. It is computationally hard to reverse the transformation or to find collisions. MD5 and SHA are examples of one-way hash functions.

RC2, RC4: Proprietary bulk ciphers from RSA Data Security, Inc. (There is no good reference to these as they are unpublished works; however, see [RSADSI]). RC2 is a block cipher and RC4 is a stream cipher.

RSA: A very widely used public key algorithm that can be used for either encryption or digital signing.

salt: Non-secret random data used to make export encryption keys resist precomputation attacks.

server: The server is the application entity that responds to requests for connections from clients. The server is passive, waiting for requests from clients.

session: An SSL session is an association between a client and a server. Sessions are created by the handshake protocol. Sessions define a set of cryptographic security parameters, which can be shared among multiple connections. Sessions are used to avoid the expensive negotiation of new security parameters for each connection.

session identifier: A session identifier is a value generated by a server that identifies a particular session.

server write key: The key used to encrypt data written by the server.

server write MAC secret: The secret data used to authenticate data written by the server.

SHA: The Secure Hash Algorithm is defined in FIPS PUB 180-1. It produces a 20-byte output [SHA].

stream cipher: An encryption algorithm that converts a key into a cryptographically strong keystream, which is then exclusive-ORed with the plaintext.

symmetric cipher: See bulk cipher.

Appendix C. CipherSuite Definitions

CipherSuite	Is Exportable	Key Exchange	Cipher	Hash
SSL_NULL_WITH_NULL_NULL	*	NULL	NULL	NULL
SSL_RSA_WITH_NULL_MD5	*	RSA	NULL	MD5
SSL_RSA_WITH_NULL_SHA	*	RSA	NULL	SHA
SSL_RSA_EXPORT_WITH_RC4_40_MD5	*	RSA_EXPORT	RC4_40	MD5
SSL_RSA_WITH_RC4_128_MD5		RSA	RC4_128	MD5
SSL_RSA_WITH_RC4_128_SHA		RSA	RC4_128	SHA
SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5	*	RSA_EXPORT	RC2_CBC_40	MD5
SSL_RSA_WITH_IDEA_CBC_SHA		RSA	IDEA_CBC	SHA
SSL_RSA_EXPORT_WITH_DES40_CBC_SHA	*	RSA_EXPORT	DES40_CBC	SHA
SSL_RSA_WITH_DES_CBC_SHA		RSA	DES_CBC	SHA
SSL_RSA_WITH_3DES_EDE_CBC_SHA		RSA	3DES_EDE_CBC	SHA
SSL_DH_DSS_EXPORT_WITH_DES40_CBC_SHA	*	DH_DSS_EXPORT	DES40_CBC	SHA
SSL_DH_DSS_WITH_DES_CBC_SHA		DH_DSS	DES_CBC	SHA
SSL_DH_DSS_WITH_3DES_EDE_CBC_SHA		DH_DSS	3DES_EDE_CBC	SHA
SSL_DH_RSA_EXPORT_WITH_DES40_CBC_SHA	*	DH_RSA_EXPORT	DES40_CBC	SHA
SSL_DH_RSA_WITH_DES_CBC_SHA		DH_RSA	DES_CBC	SHA
SSL_DH_RSA_WITH_3DES_EDE_CBC_SHA		DH_RSA	3DES_EDE_CBC	SHA
SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA	*	DHE_DSS_EXPORT	DES40_CBC	SHA
SSL_DHE_DSS_WITH_DES_CBC_SHA		DHE_DSS	DES_CBC	SHA
SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA		DHE_DSS	3DES_EDE_CBC	SHA
SSL_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA	*	DHE_RSA_EXPORT	DES40_CBC	SHA
SSL_DHE_RSA_WITH_DES_CBC_SHA		DHE_RSA	DES_CBC	SHA
SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA		DHE_RSA	3DES_EDE_CBC	SHA
SSL_DH_anon_EXPORT_WITH_RC4_40_MD5	*	DH_anon_EXPORT	RC4_40	MD5
SSL_DH_anon_WITH_RC4_128_MD5		DH_anon	RC4_128	MD5
SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA		DH_anon	DES40_CBC	SHA
SSL_DH_anon_WITH_DES_CBC_SHA		DH_anon	DES_CBC	SHA
SSL_DH_anon_WITH_3DES_EDE_CBC_SHA		DH_anon	3DES_EDE_CBC	SHA
SSL_FORTEZZA_KEA_WITH_NULL_SHA		FORTEZZA_KEA	NULL	SHA
SSL_FORTEZZA_KEA_WITH_FORTEZZA_CBC_SHA		FORTEZZA_KEA	FORTEZZA_CBC	SHA
SSL_FORTEZZA_KEA_WITH_RC4_128_SHA		FORTEZZA_KEA	RC4_128	SHA

Key Exchange Algorithm	Description	Key Size Limit
DHE_DSS	Ephemeral DH with DSS signatures	None
DHE_DSS_EXPORT	Ephemeral DH with DSS signatures	DH = 512 bits
DHE_RSA	Ephemeral DH with RSA signatures	None
DHE_RSA_EXPORT	Ephemeral DH with RSA signatures	DH = 512 bits, RSA = none
DH_anon	Anonymous DH, no signatures	None
DH_anon_EXPORT	Anonymous DH, no signatures	DH = 512 bits
DH_DSS	DH with DSS-based certificates	None
DH_DSS_EXPORT	DH with DSS-based certificates	DH = 512 bits
DH_RSA	DH with RSA-based certificates	None
DH_RSA_EXPORT	DH with RSA-based certificates	DH = 512 bits, RSA = none
FORTEZZA_KEA	FORTEZZA KEA. Details unpublished	N/A
NULL	No key exchange	N/A
RSA	RSA key exchange	None
RSA_EXPORT	RSA key exchange	RSA = 512 bits

Table 1

Key size limit: The key size limit gives the size of the largest public key that can be legally used for encryption in cipher suites that are exportable.

Cipher	Cipher Type	IsExportable	Key Material	Exp. Key Material	Effective Key Bits	IV Size	Block Size
NULL	Stream	*	0	0	0	0	N/A
FORTEZZA_CBC	Block		NA (**)	12 (**)	96 (**)	20 (**)	8
IDEA_CBC	Block		16	16	128	8	8
RC2_CBC_40	Block	*	5	16	40	8	8
RC4_40	Stream	*	5	16	40	0	N/A
RC4_128	Stream		16	16	128	0	N/A
DES40_CBC	Block	*	5	8	40	8	8
DES_CBC	Block		8	8	56	8	8
3DES_EDE_CBC	Block		24	24	168	8	8

* Indicates IsExportable is true.
** FORTEZZA uses its own key and IV generation algorithms.

Table 2

Key Material: The number of bytes from the key_block that are used for generating the write keys.

Expanded Key Material: The number of bytes actually fed into the encryption algorithm.

Effective Key Bits: How much entropy material is in the key material being fed into the encryption routines.

Hash Function	Hash Size	Padding Size
NULL	0	0
MD5	16	48
SHA	20	40

Table 3

Appendix D. Implementation Notes

The SSL protocol cannot prevent many common security mistakes. This section provides several recommendations to assist implementers.

D.1. Temporary RSA Keys

US export restrictions limit RSA keys used for encryption to 512 bits, but do not place any limit on lengths of RSA keys used for signing operations. Certificates often need to be larger than 512 bits, since 512-bit RSA keys are not secure enough for high-value transactions or for applications requiring long-term security. Some certificates are also designated signing-only, in which case they cannot be used for key exchange.

When the public key in the certificate cannot be used for encryption, the server signs a temporary RSA key, which is then exchanged. In exportable applications, the temporary RSA key should be the maximum allowable length (i.e., 512 bits). Because 512-bit RSA keys are relatively insecure, they should be changed often. For typical electronic commerce applications, it is suggested that keys be changed daily or every 500 transactions, and more often if possible. Note that while it is acceptable to use the same temporary key for multiple transactions, it must be signed each time it is used.

RSA key generation is a time-consuming process. In many cases, a low-priority process can be assigned the task of key generation. Whenever a new key is completed, the existing temporary key can be replaced with the new one.

D.2. Random Number Generation and Seeding

SSL requires a cryptographically secure pseudorandom number generator (PRNG). Care must be taken in designing and seeding PRNGs. PRNGs based on secure hash operations, most notably MD5 and/or SHA, are acceptable, but cannot provide more security than the size of the random number generator state. (For example, MD5-based PRNGs usually provide 128 bits of state.)

To estimate the amount of seed material being produced, add the number of bits of unpredictable information in each seed byte. For example, keystroke timing values taken from a PC-compatible’s 18.2 Hz timer provide 1 or 2 secure bits each, even though the total size of the counter value is 16 bits or more. To seed a 128-bit PRNG, one would thus require approximately 100 such timer values.

Note: The seeding functions in RSAREF and versions of BSAFE prior to 3.0 are order independent. For example, if 1000 seed bits are supplied, one at a time, in 1000 separate calls to the seed function, the PRNG will end up in a state that depends only on the number of 0 or 1 seed bits in the seed data (i.e., there are 1001 possible final states). Applications using BSAFE or RSAREF must take extra care to ensure proper seeding.

D.3. Certificates and Authentication

Implementations are responsible for verifying the integrity of certificates and should generally support certificate revocation messages. Certificates should always be verified to ensure proper signing by a trusted certificate authority (CA). The selection and addition of trusted CAs should be done very carefully. Users should be able to view information about the certificate and root CA.

D.4. CipherSuites

SSL supports a range of key sizes and security levels, including some that provide no or minimal security. A proper implementation will probably not support many cipher suites. For example, 40-bit encryption is easily broken, so implementations requiring strong security should not allow 40-bit keys. Similarly, anonymous Diffie-Hellman is strongly discouraged because it cannot prevent man-in-the-middle attacks. Applications should also enforce minimum and maximum key sizes. For example, certificate chains containing 512-bit RSA keys or signatures are not appropriate for high-security applications.

D.5. FORTEZZA

This section describes implementation details for cipher suites that make use of the FORTEZZA hardware encryption system.

D.5.1. Notes on Use of FORTEZZA Hardware

A complete explanation of all issues regarding the use of FORTEZZA hardware is outside the scope of this document. However, there are a few special requirements of SSL that deserve mention.

Because SSL is a full duplex protocol, two crypto states must be maintained, one for reading and one for writing. There are also a number of circumstances that can result in the crypto state in the FORTEZZA card being lost. For these reasons, it’s recommended that the current crypto state be saved after processing a record, and loaded before processing the next.

After the client generates the TEK, it also generates two message encryption keys (MEKs), one for reading and one for writing. After generating each of these keys, the client must generate a corresponding IV and then save the crypto state. The client also uses the TEK to generate an IV and encrypt the premaster secret. All three IVs are sent to the server, along with the wrapped keys and the encrypted premaster secret in the client key exchange message. At this point, the TEK is no longer needed, and may be discarded.

On the server side, the server uses the master IV and the TEK to decrypt the premaster secret. It also loads the wrapped MEKs into the card. The server loads both IVs to verify that the IVs match the keys. However, since the card is unable to encrypt after loading an IV, the server must generate a new IV for the server write key. This IV is discarded.

When encrypting the first encrypted record (and only that record), the server adds 8 bytes of random data to the beginning of the fragment. These 8 bytes are discarded by the client after decryption. The purpose of this is to synchronize the state on the client and server resulting from the different IVs.

D.5.2. FORTEZZA Cipher Suites

5) FORTEZZA_NULL_WITH_NULL_SHA: Uses the full FORTEZZA key exchange, including sending server and client write keys and IVs.

D.5.3. FORTEZZA Session Resumption

There are two possibilities for FORTEZZA session restart: 1) Never restart a FORTEZZA session. 2) Restart a session with the previously negotiated keys and IVs.

Never restarting a FORTEZZA session:

Clients who never restart FORTEZZA sessions should never send session IDs that were previously used in a FORTEZZA session as part of the ClientHello. Servers who never restart FORTEZZA sessions should never send a previous session id on the ServerHello if the negotiated session is FORTEZZA.

Restart a session:

You cannot restart FORTEZZA on a session that has never done a complete FORTEZZA key exchange (that is, you cannot restart FORTEZZA if the session was an RSA/RC4 session renegotiated for FORTEZZA). If you wish to restart a FORTEZZA session, you must save the MEKs and

IVs from the initial key exchange for this session and reuse them for any new connections on that session. This is not recommended, but it is possible.

Appendix E. Version 2.0 Backward Compatibility

Version 3.0 clients that support version 2.0 servers must send version 2.0 client hello messages [SSL-2]. Version 3.0 servers should accept either client hello format. The only deviations from the version 2.0 specification are the ability to specify a version with a value of three and the support for more ciphering types in the CipherSpec.

Warning: The ability to send version 2.0 client hello messages will be phased out with all due haste. Implementers should make every effort to move forward as quickly as possible. Version 3.0 provides better mechanisms for transitioning to newer versions.

The following cipher specifications are carryovers from SSL version 2.0. These are assumed to use RSA for key exchange and authentication.

```
V2CipherSpec SSL_RC4_128_WITH_MD5          = { 0x01,0x00,0x80 };
V2CipherSpec SSL_RC4_128_EXPORT40_WITH_MD5 = { 0x02,0x00,0x80 };
V2CipherSpec SSL_RC2_CBC_128_CBC_WITH_MD5  = { 0x03,0x00,0x80 };
V2CipherSpec SSL_RC2_CBC_128_CBC_EXPORT40_WITH_MD5
                                           = { 0x04,0x00,0x80 };
V2CipherSpec SSL_IDEA_128_CBC_WITH_MD5     = { 0x05,0x00,0x80 };
V2CipherSpec SSL_DES_64_CBC_WITH_MD5       = { 0x06,0x00,0x40 };
V2CipherSpec SSL_DES_192_EDE3_CBC_WITH_MD5 = { 0x07,0x00,0xC0 };
```

Cipher specifications introduced in version 3.0 can be included in version 2.0 client hello messages using the syntax below. Any V2CipherSpec element with its first byte equal to zero will be ignored by version 2.0 servers. Clients sending any of the above V2CipherSpecs should also include the version 3.0 equivalent (see Appendix A.6):

```
V2CipherSpec (see Version 3.0 name) = { 0x00, CipherSuite };
```

E.1. Version 2 Client Hello

The version 2.0 client hello message is presented below using this document's presentation model. The true definition is still assumed to be the SSL version 2.0 specification.

```
      uint8 V2CipherSpec[3];

      struct {
          unit8 msg_type;
          Version version;
          uint16 cipher_spec_length;
          uint16 session_id_length;
          uint16 challenge_length;
          V2CipherSpec cipher_specs[V2ClientHello.cipher_spec_length];
          opaque session_id[V2ClientHello.session_id_length];
          Random challenge;
      } V2ClientHello;
```

session msg_type: This field, in conjunction with the version field, identifies a version 2 client hello message. The value should equal one (1).

version: The highest version of the protocol supported by the client (equals ProtocolVersion.version; see Appendix A.1).

cipher_spec_length: This field is the total length of the field cipher_specs. It cannot be zero and must be a multiple of the V2CipherSpec length (3).

session_id_length: This field must have a value of either zero or 16. If zero, the client is creating a new session. If 16, the session_id field will contain the 16 bytes of session identification.

challenge_length: The length in bytes of the client's challenge to the server to authenticate itself. This value must be 32.

cipher_specs: This is a list of all CipherSpecs the client is willing and able to use. There must be at least one CipherSpec acceptable to the server.

session_id: If this field's length is not zero, it will contain the identification for a session that the client wishes to resume.

challenge: The client's challenge to the server for the server to identify itself is a (nearly) arbitrary length random. The version 3.0 server will right justify the challenge data to become the ClientHello.random data (padded with leading zeroes, if necessary), as specified in this version 3.0 protocol. If the length of the challenge is greater than 32 bytes, then only the last 32 bytes are used. It is legitimate (but not necessary) for a V3 server to reject a V2 ClientHello that has fewer than 16 bytes of challenge data.

Note: Requests to resume an SSL 3.0 session should use an SSL 3.0 client hello.

E.2. Avoiding Man-in-the-Middle Version Rollback

When SSL version 3.0 clients fall back to version 2.0 compatibility mode, they use special PKCS #1 block formatting. This is done so that version 3.0 servers will reject version 2.0 sessions with version 3.0-capable clients.

When version 3.0 clients are in version 2.0 compatibility mode, they set the right-hand (least-significant) 8 random bytes of the PKCS padding (not including the terminal null of the padding) for the RSA encryption of the ENCRYPTED-KEY-DATA field of the CLIENT-MASTER-KEY to 0x03 (the other padding bytes are random). After decrypting the ENCRYPTED-KEY-DATA field, servers that support SSL 3.0 should issue an error if these eight padding bytes are 0x03. Version 2.0 servers receiving blocks padded in this manner will proceed normally.

Appendix F. Security Analysis

The SSL protocol is designed to establish a secure connection between a client and a server communicating over an insecure channel. This document makes several traditional assumptions, including that attackers have substantial computational resources and cannot obtain secret information from sources outside the protocol. Attackers are assumed to have the ability to capture, modify, delete, replay, and otherwise tamper with messages sent over the communication channel. This appendix outlines how SSL has been designed to resist a variety of attacks.

F.1. Handshake Protocol

The handshake protocol is responsible for selecting a CipherSpec and generating a MasterSecret, which together comprise the primary cryptographic parameters associated with a secure session. The handshake protocol can also optionally authenticate parties who have certificates signed by a trusted certificate authority.

F.1.1. Authentication and Key Exchange

SSL supports three authentication modes: authentication of both parties, server authentication with an unauthenticated client, and total anonymity. Whenever the server is authenticated, the channel should be secure against man-in-the-middle attacks, but completely anonymous sessions are inherently vulnerable to such attacks.

Anonymous servers cannot authenticate clients, since the client signature in the certificate verify message may require a server certificate to bind the signature to a particular server. If the server is authenticated, its certificate message must provide a valid certificate chain leading to an acceptable certificate authority. Similarly, authenticated clients must supply an acceptable certificate to the server. Each party is responsible for verifying that the other's certificate is valid and has not expired or been revoked.

The general goal of the key exchange process is to create a pre_master_secret known to the communicating parties and not to attackers. The pre_master_secret will be used to generate the master_secret (see Section 6.1). The master_secret is required to generate the finished messages, encryption keys, and MAC secrets (see Sections 5.6.9 and 6.2.2). By sending a correct finished message, parties thus prove that they know the correct pre_master_secret.

F.1.1.1. Anonymous Key Exchange

Completely anonymous sessions can be established using RSA, Diffie-Hellman, or FORTEZZA for key exchange. With anonymous RSA, the client encrypts a pre_master_secret with the server's uncertified public key extracted from the server key exchange message. The result is sent in a client key exchange message. Since eavesdroppers do not know the server's private key, it will be infeasible for them to decode the pre_master_secret.

With Diffie-Hellman or FORTEZZA, the server's public parameters are contained in the server key exchange message and the client's are sent in the client key exchange message. Eavesdroppers who do not know the private values should not be able to find the Diffie-Hellman result (i.e., the pre_master_secret) or the FORTEZZA token encryption key (TEK).

Warning: Completely anonymous connections only provide protection against passive eavesdropping. Unless an independent tamper-proof channel is used to verify that the finished messages were not replaced by an attacker, server authentication is required in environments where active man-in-the-middle attacks are a concern.

F.1.1.2. RSA Key Exchange and Authentication

With RSA, key exchange and server authentication are combined. The public key either may be contained in the server's certificate or may be a temporary RSA key sent in a server key exchange message. When temporary RSA keys are used, they are signed by the server's RSA or DSS certificate. The signature includes the current

ClientHello.random, so old signatures and temporary keys cannot be replayed. Servers may use a single temporary RSA key for multiple negotiation sessions.

Note: The temporary RSA key option is useful if servers need large certificates but must comply with government-imposed size limits on keys used for key exchange.

After verifying the server's certificate, the client encrypts a pre_master_secret with the server's public key. By successfully decoding the pre_master_secret and producing a correct finished message, the server demonstrates that it knows the private key corresponding to the server certificate.

When RSA is used for key exchange, clients are authenticated using the certificate verify message (see Section 5.6.8). The client signs a value derived from the master_secret and all preceding handshake messages. These handshake messages include the server certificate, which binds the signature to the server, and ServerHello.random, which binds the signature to the current handshake process.

F.1.1.3. Diffie-Hellman Key Exchange with Authentication

When Diffie-Hellman key exchange is used, the server either can supply a certificate containing fixed Diffie-Hellman parameters or can use the server key exchange message to send a set of temporary Diffie-Hellman parameters signed with a DSS or RSA certificate. Temporary parameters are hashed with the hello.random values before signing to ensure that attackers do not replay old parameters. In either case, the client can verify the certificate or signature to ensure that the parameters belong to the server.

If the client has a certificate containing fixed Diffie-Hellman parameters, its certificate contains the information required to complete the key exchange. Note that in this case, the client and server will generate the same Diffie-Hellman result (i.e., pre_master_secret) every time they communicate. To prevent the pre_master_secret from staying in memory any longer than necessary, it should be converted into the master_secret as soon as possible. Client Diffie-Hellman parameters must be compatible with those supplied by the server for the key exchange to work.

If the client has a standard DSS or RSA certificate or is unauthenticated, it sends a set of temporary parameters to the server in the client key exchange message, then optionally uses a certificate verify message to authenticate itself.

F.1.1.4. FORTEZZA

FORTEZZA's design is classified, but at the protocol level it is similar to Diffie-Hellman with fixed public values contained in certificates. The result of the key exchange process is the token encryption key (TEK), which is used to wrap data encryption keys, client write key, server write key, and master secret encryption key. The data encryption keys are not derived from the pre_master_secret because unwrapped keys are not accessible outside the token. The encrypted pre_master_secret is sent to the server in a client key exchange message.

F.1.2. Version Rollback Attacks

Because SSL version 3.0 includes substantial improvements over SSL version 2.0, attackers may try to make version 3.0-capable clients and servers fall back to version 2.0. This attack is occurring if (and only if) two version 3.0-capable parties use an SSL 2.0 handshake.

Although the solution using non-random PKCS #1 block type 2 message padding is inelegant, it provides a reasonably secure way for version 3.0 servers to detect the attack. This solution is not secure against attackers who can brute force the key and substitute a new ENCRYPTED-KEY-DATA message containing the same key (but with normal padding) before the application specified wait threshold has expired. Parties concerned about attacks of this scale should not be using 40-bit encryption keys anyway. Altering the padding of the least significant 8 bytes of the PKCS padding does not impact security, since this is essentially equivalent to increasing the input block size by 8 bytes.

F.1.3. Detecting Attacks against the Handshake Protocol

An attacker might try to influence the handshake exchange to make the parties select different encryption algorithms than they would normally choose. Because many implementations will support 40-bit exportable encryption and some may even support null encryption or MAC algorithms, this attack is of particular concern.

For this attack, an attacker must actively change one or more handshake messages. If this occurs, the client and server will compute different values for the handshake message hashes. As a result, the parties will not accept each other's finished messages. Without the master_secret, the attacker cannot repair the finished messages, so the attack will be discovered.

F.1.4. Resuming Sessions

When a connection is established by resuming a session, new ClientHello.random and ServerHello.random values are hashed with the session's master_secret. Provided that the master_secret has not been compromised and that the secure hash operations used to produce the encryption keys and MAC secrets are secure, the connection should be secure and effectively independent from previous connections. Attackers cannot use known encryption keys or MAC secrets to compromise the master_secret without breaking the secure hash operations (which use both SHA and MD5).

Sessions cannot be resumed unless both the client and server agree. If either party suspects that the session may have been compromised, or that certificates may have expired or been revoked, it should force a full handshake. An upper limit of 24 hours is suggested for session ID lifetimes, since an attacker who obtains a master_secret may be able to impersonate the compromised party until the corresponding session ID is retired. Applications that may be run in relatively insecure environments should not write session IDs to stable storage.

F.1.5. MD5 and SHA

SSL uses hash functions very conservatively. Where possible, both MD5 and SHA are used in tandem to ensure that non-catastrophic flaws in one algorithm will not break the overall protocol.

F.2. Protecting Application Data

The master_secret is hashed with the ClientHello.random and ServerHello.random to produce unique data encryption keys and MAC secrets for each connection. FORTEZZA encryption keys are generated by the token, and are not derived from the master_secret.

Outgoing data is protected with a MAC before transmission. To prevent message replay or modification attacks, the MAC is computed from the MAC secret, the sequence number, the message length, the message contents, and two fixed-character strings. The message type field is necessary to ensure that messages intended for one SSL record layer client are not redirected to another. The sequence number ensures that attempts to delete or reorder messages will be detected. Since sequence numbers are 64 bits long, they should never overflow. Messages from one party cannot be inserted into the other's output, since they use independent MAC secrets. Similarly, the server-write and client-write keys are independent so stream cipher keys are used only once.

If an attacker does break an encryption key, all messages encrypted with it can be read. Similarly, compromise of a MAC key can make message modification attacks possible. Because MACs are also encrypted, message-alteration attacks generally require breaking the encryption algorithm as well as the MAC.

Note: MAC secrets may be larger than encryption keys, so messages can remain tamper resistant even if encryption keys are broken.

F.3. Final Notes

For SSL to be able to provide a secure connection, both the client and server systems, keys, and applications must be secure. In addition, the implementation must be free of security errors.

The system is only as strong as the weakest key exchange and authentication algorithm supported, and only trustworthy cryptographic functions should be used. Short public keys, 40-bit bulk encryption keys, and anonymous servers should be used with great caution. Implementations and users must be careful when deciding which certificates and certificate authorities are acceptable; a dishonest certificate authority can do tremendous damage.

Appendix G. Acknowledgements

G.1. Other Contributors

Martin Abadi
Digital Equipment Corporation
ma@pa.dec.com

Robert Relyea
Netscape Communications
relyea@netscape.com

Taher Elgamal
Netscape Communications
elgamal@netscape.com

Jim Roskind
Netscape Communications
jar@netscape.com

Anil Gangolli
Netscape Communications
gangolli@netscape.com

Micheal J. Sabin, Ph.D.
Consulting Engineer
msabin@netcom.com

Kipp E.B. Hickman
Netscape Communications
kipp@netscape.com

Tom Weinstein
Netscape Communications
tomw@netscape.com

G.2. Early Reviewers

Robert Baldwin
RSA Data Security, Inc.
baldwin@rsa.com

Clyde Monma
Bellcore
clyde@bellcore.com

George Cox
Intel Corporation
cox@ibeam.jf.intel.com

Eric Murray
ericm@lne.com

Cheri Dowell
Sun Microsystems
cheri@eng.sun.com

Avi Rubin
Bellcore
rubin@bellcore.com

Stuart Haber
Bellcore
stuart@bellcore.com

Don Stephenson
Sun Microsystems
don.stephenson@eng.sun.com

Burt Kaliski
RSA Data Security, Inc.
burt@rsa.com

Joe Tardo
General Magic
tardo@genmagic.com

Authors' Addresses

Alan O. Freier
Netscape Communications

Philip Karlton
Netscape Communications

Paul C. Kocher
Independent Consultant