Network Working Group
Internet-Draft
Expires: March 31, 2004

T. Ylonen
SSH Communications Security Corp
D. Moffat, Editor, Ed.
Sun Microsystems, Inc
Oct 2003

SSH Transport Layer Protocol
draft-ietf-secsh-transport-17.txt

Status of this Memo

Copyright Notice

Abstract

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. Contributors

The major original contributors of this document were: Tatu Ylonen, Tero Kivinen, Timo J. Rinne, Sami Lehtinen (all of SSH Communications Security Corp), and Markku-Juhani O. Saarinen (University of Jyvaskyla)

The document editor is: Darren.Moffat@Sun.COM. Comments on this internet draft should be sent to the IETF SECSH working group, details at: http://ietf.org/html.charters/secsh-charter.html

2. 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, flexible, to allow parameter negotiation, and to minimize the number of round-trips. 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.

3. Conventions Used in This Document

The keywords "MUST", "MUST NOT", "REQUIRED", "SHOULD", "SHOULD NOT", and "MAY" that appear in this document are to be interpreted as described in [RFC2119].

The used data types and terminology are specified in the architecture document [SSH-ARCH].

The architecture document also discusses the algorithm naming conventions that MUST be used with the SSH protocols.

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 of the form "SSH-protoversion-softwareversion comments", followed by carriage return and newline characters (ASCII 13 and 10, respectively). Both sides MUST be able to process identification strings without carriage return character. No null character is sent. The maximum length of the string is 255 characters, including the carriage return and newline.

The part of the identification string preceding carriage return and newline is used in the Diffie-Hellman key exchange (see Section Section 7).

The server MAY send other lines of data before sending the version string. Each line SHOULD be terminated by a carriage return and newline. Such lines MUST NOT begin with "SSH-", and SHOULD be encoded in ISO-10646 UTF-8 [RFC2279] (language is not specified). Clients MUST be able to process such lines; they MAY be silently ignored, or MAY be displayed to the client user; if they are displayed, control character filtering 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.

Version strings MUST consist of printable US-ASCII characters, not including whitespaces or a minus sign (-). The version string is primarily used to trigger compatibility extensions and to indicate the capabilities of an implementation. The comment string should contain additional information that might be useful in solving user problems.

The protocol version described in this document is 2.0.

Key exchange will begin immediately after sending this identifier. All packets following the identification string SHALL use the binary packet protocol, to be described below.

4.3 Compatibility With Old SSH 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 an older version of the protocol. Information in this section is only relevant for implementations supporting compatibility with SSH versions 1.x. There is no standards track or informational draft available that defines the SSH 1.x protocol. The only known documentation of the 1.x protocol is contained in README files that are shipped along with the source code.

4.3.1 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 protocol version 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 version identification string.

In the compatibility mode the server SHOULD NOT send any further data after its initialization 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 version string.

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

4.3.2 New Client, Old Server

Since the new client MAY immediately send additional data after its identification string (before receiving server's identification), the old protocol may already have been corrupted 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. 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); m = mac_length
```

packet_length

The length of the packet (bytes), not including MAC or the packet_length field itself.

padding_length

Length of 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 || 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 length of the concatenation of packet length, padding length, payload, and 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.

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.

5.1 Maximum Packet Length

All implementations MUST be able to process packets with uncompressed payload length of 32768 bytes or less and total packet size of 35000 bytes or less (including length, padding length, payload, padding, and MAC.). The maximum of 35000 bytes is an arbitrary chosen value larger than uncompressed size. Implementations SHOULD support longer packets, where they might be needed, e.g. if an implementation wants to send a very large number of certificates. Such packets MAY be sent if the version string indicates that the other party is able to process them. However, implementations SHOULD check that the packet length is reasonable for the implementation to avoid denial-of-service and/or buffer overflow attacks.

5.2 Compression

If compression has been negotiated, the payload field (and only it) will be compressed using the negotiated algorithm. The 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 independently choosing the algorithm for each direction.

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].

5.3 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, and implementations MUST allow independently choosing the algorithm for each direction (if multiple algorithms are allowed by local policy).

The following ciphers are currently defined:

3des-cbc	REQUIRED	three-key 3DES in CBC mode
blowfish-cbc	OPTIONALi	Blowfish in CBC mode
twofish256-cbc	OPTIONAL	Twofish in CBC mode, with 256-bit key
twofish-cbc	OPTIONAL	alias for "twofish256-cbc" (this is being retained for historical reasons)
twofish192-cbc	OPTIONAL	Twofish with 192-bit key
twofish128-cbc	OPTIONAL	Twofish with 128-bit key
aes256-cbc	OPTIONAL	AES (Rijndael) in CBC mode, with 256-bit key
aes192-cbc	OPTIONAL	AES with 192-bit key
aes128-cbc	RECOMMENDED	AES with 128-bit key
serpent256-cbc	OPTIONAL	Serpent in CBC mode, with 256-bit key
serpent192-cbc	OPTIONAL	Serpent with 192-bit key
serpent128-cbc	OPTIONAL	Serpent with 128-bit key
arcfour	OPTIONAL	the ARCFOUR stream cipher
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]

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. Same as above but with 192-bit key.

The "twofish128-cbc" cipher. Same as above but with 128-bit key.

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

The "aes192-cbc" cipher. Same as above but with 192-bit key.

The "aes128-cbc" cipher. Same as above but with 128-bit key.

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

The "serpent192-cbc" cipher. Same as above but with 192-bit key.

The "serpent128-cbc" cipher. Same as above but with 128-bit key.

The "arcfour" is the Arcfour stream cipher with 128 bit keys. The Arcfour cipher is believed to be compatible with the RC4 cipher [SCHNEIER]. RC4 is a registered trademark of RSA Data Security Inc. 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 [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].

5.4 Data Integrity

Data integrity is protected by including with each packet a message authentication code (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 selected MAC 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 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.

The MAC bytes 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

Figure 1

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

The hash algorithms are described in [SCHNEIER].

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

5.5 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.

Only one REQUIRED key exchange method has been defined:

diffie-hellman-group1-sha1 REQUIRED

This method is described later in this document.

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

5.6 Public Key Algorithms

This protocol has been designed to be able 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 in e.g. 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
x509v3-sign-rsa	OPTIONAL	sign	X.509 certificates (RSA key)
x509v3-sign-dss	OPTIONAL	sign	X.509 certificates (DSS key)
spki-sign-rsa	OPTIONAL	sign	SPKI certificates (RSA key)
spki-sign-dss	OPTIONAL	sign	SPKI certificates (DSS 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].

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 have 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 / certifcate 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 / cert 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] using the SHA-1 hash. A description can also be found in [SCHNEIER].

The resulting signature is encoded as follows:

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

dss_signature_blob is encoded as a string containing r followed by s (which are 160 bits long 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 done according to [SCHNEIER] and [PKCS1] using the SHA-1 hash.

The resulting signature is encoded as follows:

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

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 "spki-sign-rsa" method indicates that the certificate blob contains a sequence of SPKI certificates. The format of SPKI certificates is described in [RFC2693]. This method indicates that the key (or one of the keys in the certificate) is an RSA-key.

The "spki-sign-dss". As above, but indicates that the key (or one of the keys in the certificate) is a DSS-key.

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". As above, but indicates that the key is a DSS-key.

6. Key Exchange

Key exchange begins by each side sending 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.

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 Section 6.1).

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.

Server authentication in the key exchange MAY be implicit. After a key exchange with implicit server authentication, the client MUST wait for response to its service request message before sending any further data.

6.1 Algorithm Negotiation

Key exchange begins by each side sending the following packet:

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

Each of the algorithm strings MUST be a comma-separated list of algorithm names (see ''Algorithm Naming'' in [SSH-ARCH]). Each supported (allowed) algorithm MUST be listed in order of preference.

The first algorithm in each list MUST be the preferred (guessed) algorithm. Each string 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

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 encryption capable host key. It MUST be possible to determine this from the public key algorithm name. The first algorithm on the client's 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

Lists the acceptable symmetric encryption algorithms in order of preference. The chosen encryption algorithm to each direction MUST be the first algorithm on the client's list that is also on the server's 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 Section 5.3.

mac_algorithms

Lists the acceptable MAC algorithms in order of preference. The chosen MAC algorithm MUST be the first algorithm on the client's list that is also on the server's 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 Figure 1.

compression_algorithms

Lists the acceptable compression algorithms in order of preference. The chosen compression algorithm MUST be the first algorithm on the client's list that is also on the server's 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 Section 5.2.

languages

This is a comma-separated list of language tags in order of preference [RFC3066]. Both parties MAY ignore this list. If there are no language preferences, this list SHOULD be empty. 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 KEXINIT packet exchange, the key exchange algorithm is run. It may involve several packet exchanges, as specified by the key exchange method.

6.2 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:

- o 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).
- o Initial IV server to client: HASH(K || H || "B" || session_id)
- o Encryption key client to server: HASH(K || H || "C" || session_id)
- o Encryption key server to client: HASH(K || H || "D" || session_id)
- o Integrity key client to server: HASH(K || H || "E" || session_id)
- o Integrity key server to client: HASH(K || H || "F" || session_id)

Key data MUST be taken from the beginning of the hash output. 128 bits (16 bytes) MUST be used for algorithms with variable-length keys. The only variable key length algorithm defined in this document is arcfour). For other algorithms, as many bytes as are 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.

6.3 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 taken into use for receiving.

This message is the only valid message after key exchange, in addition to SSH_MSG_DEBUG, SSH_MSG_DISCONNECT and SSH_MSG_IGNORE messages. The purpose of this message is to ensure that a party is able to respond with a disconnect message that the other party can understand if something goes wrong with the key exchange. Implementations MUST NOT accept any other messages after key exchange before receiving SSH_MSG_NEWKEYS.

```
  byte      SSH_MSG_NEWKEYS
```

7. Diffie-Hellman Key Exchange

The Diffie-Hellman key exchange provides a shared secret that can not be determined by either party alone. The key exchange is combined with a signature with the host key to provide host authentication.

In the following description (C is the client, S is the server; p is a large safe prime, g is a generator for a subgroup of GF(p), and q is the order of the subgroup; V_S is S's version string; V_C is C's version string; K_S is S's public host key; I_C is C's KEXINIT message and I_S S's KEXINIT message which 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.

Either side MUST NOT send or accept e or f values that are not in the range [1, p-1]. 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 KEXINIT messages.

First, the client sends the following:

```
byte      SSH_MSG_KEXDH_INIT
mpint     e
```

The server 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 version string (CR and NL excluded)
string    V_S, the server's version string (CR and NL 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. For example, "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.

7.1 diffie-hellman-group1-sha1

The "diffie-hellman-group1-sha1" method specifies Diffie-Hellman key exchange with SHA-1 as HASH, and Oakley group 14 [RFC3526] (2048-bit MODP Group). It is included below in hexadecimal and decimal.

The prime p is equal to 2^1024 - 2^960 - 1 + 2^64 * floor(2^894 Pi + 129093). Its hexadecimal value is:

```
FFFFFFFF FFFFFFFF C90FDAA2 2168C234 C4C6628B 80DC1CD1
29024E08 8A67CC74 020BBEA6 3B139B22 514A0879 8E3404DD
EF9519B3 CD3A431B 302B0A6D F25F1437 4FE1356D 6D51C245
E485B576 625E7EC6 F44C42E9 A637ED6B 0BFF5CB6 F406B7ED
EE386BFB 5A899FA5 AE9F2411 7C4B1FE6 49286651 ECE65381
FFFFFFFF FFFFFFFF.
```

In decimal, this value is:

```
179769313486231590770839156793787453197860296048756011706444
423684197180216158519368947833795864925541502180565485980503
646440548199239100050792877003355816639229553136239076508735
759914822574862575007425302077447712589550957937778424442426
617334727629299387668709205606050270810842907692932019128194
467627007.
```

The generator used with this prime is g = 2. The group order q is (p - 1) / 2.

8. 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 Section 6.1). 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 are 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.

9. 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].

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 "servicename@domain" syntax.

```
  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 Section 6 in [SSH-ARCH]). 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 response to its service request message before sending any further data.

10. Additional Messages

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

10.1 Disconnection Message

```
  byte      SSH_MSG_DISCONNECT
  uint32    reason code
  string    description [RFC2279]
  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 description field gives a more specific explanation in a human-readable form. The error code gives the reason in a more machine-readable format (suitable for localization), and can have the following values:

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

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

10.2 Ignored Data Message

```
  byte      SSH_MSG_IGNORE
  string    data
```

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

10.3 Debug Message

```
  byte      SSH_MSG_DEBUG
  boolean   always_display
  string    message [RFC2279]
  string    language tag [RFC3066]
```

All implementations MUST understand this message, but they are allowed to ignore it. This message is used to pass the other side 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, terminal control character filtering discussed in [SSH-ARCH] should be used to avoid attacks by sending terminal control characters.

10.4 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
```

11. Summary of Message Numbers

The following message numbers have been defined in this protocol:

```
  #define SSH_MSG_DISCONNECT             1
  #define SSH_MSG_IGNORE                 2
  #define SSH_MSG_UNIMPLEMENTED          3
  #define SSH_MSG_DEBUG                  4
  #define SSH_MSG_SERVICE_REQUEST        5
  #define SSH_MSG_SERVICE_ACCEPT         6

  #define SSH_MSG_KEXINIT                20
  #define SSH_MSG_NEWKEYS                21
```

```
  /* Numbers 30-49 used for kex packets.
     Different kex methods may reuse message numbers in
     this range. */

  #define SSH_MSG_KEXDH_INIT             30
  #define SSH_MSG_KEXDH_REPLY            31
```

12. IANA Considerations

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

13. 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 Section 8 of [SSH-ARCH]

14. Intellectual Property

The IETF takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Information on the IETF’s procedures with respect to rights in standards-track and standards-related documentation can be found in BCP-11. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this specification can be obtained from the IETF Secretariat.

The IETF has been notified of intellectual property rights claimed in regard to some or all of the specification contained in this document. For more information consult the online list of claimed rights.

15. Additional Information

The current document editor is: Darren.Moffat@Sun.COM. Comments on this internet draft should be sent to the IETF SECSH working group, details at: http://ietf.org/html.charters/secsh-charter.html

Normative

[SSH-ARCH] Ylonen, T., "SSH Protocol Architecture", I-D draft-ietf-architecture-15.txt, Oct 2003.

[SSH-TRANS] Ylonen, T., "SSH Transport Layer Protocol", I-D draft-ietf-transport-17.txt, Oct 2003.

[SSH-USERAUTH] Ylonen, T., "SSH Authentication Protocol", I-D draft-ietf-userauth-18.txt, Oct 2003.

[SSH-CONNECT] Ylonen, T., "SSH Connection Protocol", I-D draft-ietf-connect-18.txt, Oct 2003.

[SSH-NUMBERS] Lehtinen, S. and D. Moffat, "SSH Protocol Assigned Numbers", I-D draft-ietf-secsh-assignednumbers-05.txt, Oct 2003.

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

Informative

[FIPS-186] Federal Information Processing Standards Publication, "FIPS PUB 186, Digital Signature Standard", May 1994.

[FIPS-197] NIST, "FIPS PUB 197 Advanced Encryption Standard (AES)", November 2001.

[FIPS-46-3] U.S. Dept. of Commerce, "FIPS PUB 46-3, Data Encryption Standard (DES)", October 1999.

[RFC2459] Housley, R., Ford, W., Polk, T. and D. Solo, "Internet X.509 Public Key Infrastructure Certificate and CRL Profile", RFC 2459, January 1999.

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

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

[RFC1950] Deutsch, L. 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.

[RFC2279] Yergeau, F., "UTF-8, a transformation format of ISO 10646", RFC 2279, January 1998.

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

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

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

[RFC2693] Ellison, C., Frantz, B., Lampson, B., Rivest, R., Thomas, B. and T. Ylonen, "SPKI Certificate Theory", RFC 2693, September 1999.

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

[SCHNEIER] Schneier, B., "Applied Cryptography Second Edition: protocols algorithms and source in code in C", 1996.

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

Authors' Addresses

Tatu Ylonen
SSH Communications Security Corp
Fredrikinkatu 42
HELSINKI FIN-00100
Finland

EMail: ylo@ssh.com

Darren J. Moffat (editor)
Sun Microsystems, Inc
17 Network Circle
Menlo Park 95025
USA

EMail: Darren.Moffat@Sun.COM

Appendix A. Contibutors

Intellectual Property Statement

The IETF takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Information on the IETF's procedures with respect to rights in standards-track and standards-related documentation can be found in BCP-11. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementors or users of this specification can be obtained from the IETF Secretariat.

The IETF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights which may cover technology that may be required to practice this standard. Please address the information to the IETF Executive Director.

The IETF has been notified of intellectual property rights claimed in regard to some or all of the specification contained in this document. For more information consult the online list of claimed rights.

Full Copyright Statement

Copyright (C) The Internet Society (2003). All Rights Reserved.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this paragraph are included on all such copies and derivative works. However, this document itself may not be modified in any way, such as by removing the copyright notice or references to the Internet Society or other Internet organizations, except as needed for the purpose of developing Internet standards in which case the procedures for copyrights defined in the Internet Standards process must be followed, or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by the Internet Society or its successors or assignees.

Acknowledgment

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