Network Working Group
INTERNET-DRAFT
draft-ietf-tls-ssh-00.txt
Expires: December 1st, 1996

Tatu Ylonen <ylo@ssh.fi>
SSH Communications Security
June 13, 1996

SSH Transport Layer Protocol

Status of This memo

Abstract

This document describes the SSH transport layer protocol. The protocol can be used as a basis for a number of secure network services. It provides strong encryption, mutual authentication, and integrity protection.

Table of Contents

1. Introduction 2
2. Data Type Representations Used in the Protocol 3
2.1. vlint32 3
2.2. string 3
2.3. boolean 3
2.4. byte 3
2.5. uint16 4
2.6. uint32 4
3. Connection Setup 4
3.1. Use over TCP/IP 4
3.2. Protocol Version Exchange 4
3.3. Compatibility with Old SSH Versions 4
3.3.1. Old Client, New Server 4
3.3.2. New Client, Old Server 5
4. Binary Packet Protocol 5
4.1. Maximum Packet Length 5
4.2. Compression 6
4.3. Encryption 6
4.4. Data Integrity 7
5. Key Exchange 7
5.1. Sending Supported Algorithm Lists 8
5.2. RSA-Style Key Exchange 11
5.2.1. Server Sends Host Key 11
5.2.2. Client Sends Double-Encrypted Session Key 11
5.3. Diffie-Hellman Style Key Exchange 12
5.4. Deriving Encryption and Integrity Keys 12
6. Client Host Authentication 13
7. Service Request 14
8. Data Exchange 14
9. Key Re-Exchange 15
10. Additional Messages 16
10.1. Disconnection Message 16
10.2. Ignored Data Message 16
10.3. Reserved Messages 16
11. Summary of Message Numbers 17
12. Public Keys and Public Key Infrastructure 17

12.1. SSH_PK_SIMPLE_RSA_PKCS 17

1. Introduction

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

Authentication in this protocol level is host-based; this protocol does not perform user authentication. It is expected that a higher level protocol will be defined on top of this protocol that will perform user authentication for those services that need it.

The protocol has been designed to be simple, flexible, allow parameter negotiation, and 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 1.5 round-trips will be needed for full key exchange, mutual authentication, service request, and acceptance notification of service request. The worst case is 2.5 round-trips for a proper implementation.

2. Data Type Representations Used in the Protocol

2.1. vlint32

The vlint32 can represent arbitrary 32-bit unsigned integers. It is stored as a variable number of bytes (1-5 bytes), depending on the value being stored.

Bits 6-7 of the first byte determine the number of additional bytes that follow, and are interpreted as follows.

Bit7	Bit6	Number of bytes that follow
0	0	0
0	1	1
1	0	2
1	1	4

Bits 0-6 of the first byte and the following bytes contain the value of the integer, MSB first.

If bits 6-7 are both one, the remaining bits in the first byte are zero (reserved for future extension).

2.2. string

A string here means an arbitrary length binary string. Strings are allowed to contain arbitrary binary data, including null characters and 8-bit characters.

A string is represented as a vlint32 containing its length, followed by zero or more characters that are the value of the string.

2.3. boolean

A boolean value is represented as a single byte. The value 0 represents false, and the value 1 represents true. All non-zero values are interpreted as true, but applications should not store values other than 0 and 1.

2.4. byte

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

2.5. uint16

A 16-bit unsigned integer, represented as two bytes, MSB first.

2.6. uint32

A 32-bit unsigned integer, represented as four bytes, MSB first.

3. Connection Setup

SSH works over any 8-bit clean, binary-transparent transport. The client initiates the connection, and sets up the binary-transparent transport.

3.1. 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 (Internet

Assigned Numbers Authority), and has been officially assigned for SSH.

3.2. Protocol Version Exchange

When the connection has been established, both sides send an identification string of the form "SSH-protoversion-softwareversion comments", followed by a newline. The maximum length of the string is 255 characters, including the newline. The protocol version described in this document is 2.0.

Key exchange will begin immediately after sending this identifier (normally without waiting for the identifier from the other side -- see the next section for compatibility issues). All packets following the identification string will use the binary packet protocol, to be described below.

3.3. Compatibility with Old SSH Versions

During a transition period, it is important to be able to work compatibly with installed SSH clients and servers using an older version of the protocol. Information in this section is only relevant for implementations supporting compatibility with old versions.

3.3.1. Old Client, New Server

Server implementations should support a configurable "compatibility" flag that enables compatibility with old versions. When this flag is on, the server will 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 desired.

When compatibility with old clients is not needed, the server should send its initial key exchange data immediately after the identification string. This saves a round-trip.

3.3.2. New Client, Old Server

Since the new client will immediately send additional data after its identification string (before receiving server's identification), the old protocol has already 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 this time.

4. Binary Packet Protocol

Each packet consists of the following fields:

Length
  The length of the packet (bytes). This represents the number of bytes that follow this value, including the optional MAC. The length is represented as a vlint32.

Padding length
  Length of padding (bytes). This field is represented as a vlint32.

Payload
  The useful contents of the packet.

Padding
  Arbitrary-length padding, such that the total length of length+paddinglength+payload+padding is a multiple of 8 bytes. It is recommended that at least four bytes of random padding be always used.

MAC
  Message authentication code. This field is optional, and its length depends on the algorithm in use.

Note that length of the concatenation of packet length, padding length, payload, and padding must be a multiple of 8. This constraint is enforced even when using stream ciphers. Note that the packet length field is also encrypted, and processing it requires special care when sending/receiving packets. In particular, one has to be extra careful when computing the amount of padding, as changing the amount of padding can also change the size of the length fields. The minimum size of a packet is 8 characters (plus MAC); implementations should decrypt the length after receiving the first 8 bytes of a packet.

When the protocol starts, no encryption is in effect, no compression is used, and no MAC is in use. During key exchange, an encryption method, compression method, and a MAC method are selected. Any further messages will use the negotiated algorithms.

4.1. Maximum Packet Length

The maximum length of the uncompressed payload is 32768 bytes. The

maximum size of the entire packet, including length, padding length, payload, padding, and MAC, is 35000 bytes. The motivation for this limit is to keep the protocol easy to implement on 16-bit machines.

4.2. Compression

If compression has been negotiated, the payload field will be compressed using the negotiated algorithm. The length field will contain the compressed length (i.e., that transmitted on the wire).

Compressed packets must not exceed the total packet size limit; the compression algorithm must guarantee that it does not expand the packet too much. The uncompressed payload size must not exceed the maximum payload size (the compressed payload, however, may be bigger than the maximum payload size, as long as the packet size limit is not exceeded).

The following compression methods are currently supported:

```
#define SSH_COMPRESS_NONE  0 /* no compression */
#define SSH_COMPRESS_ZIP   1 /* ZIP compression */
```

SSH_COMPRESS_ZIP is the ZLIB compression. Its data format is described in the Internet-Draft draft-deutsch-zlib-spec-04.txt and its references. Compression level 6 is recommended.

The compression context is initialized after 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 all data will be output, but the next packet will continue using compression tables from the end of the previous packet.

Compression is independent in each direction, and the different compression methods may be used for each direction.

4.3. Encryption

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

The encrypted data in all packets sent in one direction will be considered a single data stream. For example, initialization vectors will be passed from the end of one packet to the beginning of the next packet.

The ciphers in each direction will run independently of each other. They will typically use a different key, and different ciphers can be used in each direction.

The following ciphers/mode combinations are currently supported:

```
#define SSH_CALG_NONE        0 /* No encryption */
#define SSH_CALG_IDEA_CBC    1 /* IDEA in CBC mode */
#define SSH_CALG_3DES_CBC    2 /* 3DES in CBC mode */
```



```
#define SSH_CALG_DES_CBC     3 /* DES in CBC mode (warning: weak!) */
#define SSH_CALG_ARCFOUR     4 /* ARCFOUR stream cipher */
```

The ARCFOUR cipher is compatible with the RC4 cipher; RC4 is a trademark of RSA Data Security, Inc.

4.4. 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 will be zero. After key exchange, the selected MAC will be computed before encryption from the concatenation of packet data (length, payload, and padding) and a packet sequence number (stored as a 32-bit integer, MSB first). The integrity key is also used in the computation of the MAC, but the way it is used depends on the MAC algorithm in use. Note that the MAC algorithm may be different for each direction.

The packet sequence number is only used for integrity checking. It is never explicitly transmitted, but it is included in MAC computation to ensure that no packets are lost. The sequence number of the first packet sent is zero; from there on the sequence number is incremented by one for every packet sent (separately for each direction). The packet number is 32 bits and wraps around if 32 bits is not enough for representing it. The sequence number is incremented also for packets that are not encrypted or MACed, and is not reset even if keys are renegotiated later.

The check bytes resulting from the MAC algorithm are transmitted without encryption as the last part of the packet. The number of check bytes depends on the algorithm chosen.

The following MAC algorithms are currently defined:

```
#define SSH_MAC_NONE         0 /* No MAC in use (length = 0) */
```

```
#define SSH_MAC_HMAC_MD5     1 /* HMAC-MD5 (length = 16) */
#define SSH_MAC_HMAC_SHA     2 /* HMAC-SHA-1 (length = 20) */
#define SSH_MAC_MD5          3 /* MD5 of data+key (length = 16) */
```

The HMAC methods are described in draft-ietf-ipsec-hmac-md5-00.txt.

The SSH_MAC_MD5 method returns the MD5 of the concatenation of the authenticated data and the key.

5. Key Exchange

Key exchange begins by each side sending lists of supported algorithms. Each side has a preferred algorithm, and it is assumed that most implementations at any given time will use the same preferred algorithm. Each side will make the guess that the other side is using the same

algorithm, and may send an initial key exchange packet according to the algorithm if the preferred method so dictates. If the guess is wrong, they'll ignore each other's first data, select a common algorithm, and send the initial key exchange packet again, this time for the same algorithm.

At this time, there are two styles of key exchanges, RSA-style exchange and Diffie-Hellman style exchange (several methods may be available in each group). RSA-style exchange is based on the server sending two encryption public keys (host key and server key), the client generating a session key, sending it encrypted by the public keys, and server decrypting it with private keys. In Diffie-Hellman style exchange the parties derive a shared secret via a message exchange, and then check against man-in-the-middle attacks with a signature of the exchange.

Currently, the following key exchange methods have been defined:

```
#define SSH_KEX_RSA_SHA 1 /* RSA key exchange with SHA-1 */
#define SSH_KEX_DH_SHA  2 /* Diffie-Hellman kex with SHA-1 */
```

5.1. Sending Supported Algorithm Lists

Each side sends the following packet (this is the part that goes inside the payload):

```
vlint32   SSH_MSG_KEXINIT
byte[16]  cookie (random bytes)
string    kex_algorithms
```

```
string    host_key_algorithms
string    public_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    hash_algorithms
boolean   first_kex_packet_follows
byte[4]   0 (reserved for future extension)
```

Each of the algorithms strings contains algorithm numbers, one per byte. Each supported (allowed) algorithm should be listed, in order of preference. Each string must contain at least one value. The value "NONE" is not automatically allowed; if a party permits connections with NONE as one of the algorithms, it should list that as an algorithm.

cookie
: The cookies are random values generated by each side. They are used when deriving keys from the shared secret. Their purpose is to make it impossible for either side to fully determine the keys (which might open possibilities for passing certain signatures/authentications to third parties).

kex_algorithms
: Key exchange algorithms were defined above. The first algorithm is the preferred (and guessed) algorithm. If both sides make the same guess, that algorithm is used. Otherwise, the following algorithm is 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: 1) the server also supports the algorithm 2) if the algorithm requires an encryption-capable host key, there is an encryption-capable algorithm on the server's host_key_algorithms that is also supported by the client 3) if the algorithm requires a signature-capable host key, there is a signature-capable algorithm on the server's host_key_algorithms that is also supported by the client. If no algorithm satisfying all these conditions can be found, connection fails.

  The available algorithm numbers were listed above.

host_key_algorithms

Lists the public key algorithms/formats for which the host has a valid host key. (There can be multiple host keys for a host, possibly with different algorithm.) 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. Algorithms listed in this field must also be present in public_key_algorithms.

See Section ``Public Key Formats'' for information on the algorithm numbers.

public_key_algorithms
Lists the public key algorithms supported by the host. See Section ``Public Key Formats'' for information on the algorithm numbers.

encryption_algorithms
Lists the acceptable symmetric encryption algorithms in order of preference. The chosen encryption algorithm will be the first algorithm on the client's list that is also on the server's list. If there is no such algorithm, connection fails.

Note that SSH_CALG_NONE must be explicitly listed if it is to be acceptable. The available algorithm numbers are listed in Section ``Encryption''.

The algorithm to use is negotiated separately for each direction, and different algorithms may be chosen.

mac_algorithms
Lists the acceptable MAC algorithms in order of preference. The chosen MAC algorithm will be the first algorithm on the client's list that is also on the server's list. If there is no such algorithm, connection fails.

Note that SSH_MAC_NONE must be explicitly listed if it is to be acceptable. The available MAC algorithm numbers are listed in Section ``Data Integrity''.

The algorithm to use is negotiated separately for each direction, and different algorithms may be chosen.

compression_algorithms

Lists the acceptable compression algorithms in order of preference. The chosen compression algorithm will be the first algorithm on the client's list that is also on the server's list. If there is no such algorithm, connection fails.

Note that SSH_COMPRESS_NONE must be explicitly listed if it is to be acceptable. The available compression algorithm numbers are listed in Section ``Compression''.

The algorithm to use is negotiated separately for each direction, and different algorithms may be chosen.

hash_algorithms
Lists the acceptable hash algorithms in order of preference. The chosen hash algorithm will be the first algorithm on the client's list that is also on the server's list. If there is no such algorithm, connection fails.

Implementations should only permit algorithms that they consider to be fairly secure, as the hash function will be used e.g. for deriving various keys from the shared secret. All hash algorithms must produce at least 16 bytes of output.

Currently, the following hash functions are defined:

```
        #define SSH_HASH_SHA        1 /* returns 20 bytes */
        #define SSH_HASH_MD5        2 /* returns 16 bytes */
```

first_kex_packet_follows
Each side makes a guess of the negotiated key exchange method. This is based on the assumption that at any particular time there will be a single key exchange method and host key algorithm combination that dominates the installed base. Making a guess about the algorithm will save a round-trip in the typical case, and will incur little extra cost in the other cases.

Each side will determine if they are supposed to send an initial packet in their guessed key exchange method. If they are, they will set this field to true and follow this packet by the first key exchange packet.

After receiving the SSH_MSG_KEXINIT packet from the other side, each party will know whether their guess was right. If the guess was wrong, and this field is true, the next packet will be silently ignored, and each side will then act as determined by the

negotiated key exchange method. If the guess was right, key exchange will immediately continue.

5.2. RSA-Style Key Exchange

RSA-style key exchange requires that the server host key supports encryption. The idea is that the server sends its public host key and a periodically changing key (called the server key). The client then verifies that it is the correct key for the server, generates a session key, encrypts the session key using both the server host key and the server key, and sends the encrypted session key to the server.

The server key and host keys must both support encryption, and their sizes must be compatible in such a way that the result of encrypting a value with one of them can be encrypted with the other. The smaller of the keys must be able to encrypt at least 48 bytes.

The host key should be larger than the server key, because this causes the server key encryption to be done first, and prevents an outside attacker from replacing the outer layer of encryption by an active man-in-the-middle attack. Such an attack would not directly compromise security, but would allow the attacker to later decrypt intercepted sessions if he somehow obtains the private host key.

5.2.1. Server Sends Host Key

First, the server sends its public host and server keys in the following packet:

```
vlint32   SSH_MSG_KEXRSA_HOSTKEY
string    public host key
string    public server key
```

The host key and server key are stored in binary representation as described in Section ``Public Key Formats''.

5.2.2. Client Sends Double-Encrypted Session Key

After receiving the public keys, the client validates that the host key really belongs to the intended server. How this verification happens is not specified in this protocol. Currently it may be checked against a database of known name-key mappings; in future it will probably be validated using an Internet public key infrastructure.

If the client is not willing to trust the server host key, it simply closes the connection and the protocol terminates.

Otherwise, the client chooses an encryption algorithm that is supported

by both parties. It also chooses a MAC algorithm and a compression algorithm that are supported by both parties.

To authenticate that no-one has been manipulating the key exchange with the server, the client also computes an SHA-1 hash of the concatenated

payloads of (in this order): client's SSH_MSG_KEXINIT, server's SSH_MSG_KEXINIT, and server's SSH_MSG_KEXRSA_HOSTKEY message.

The client then generates a 256 bit random session key. A message to be passed to the server is formed by concatenating the following (in this order): six zero bytes (reserved for future), first 10 bytes of the SHA-1 hash of the key exchange, session key. This results in a total of 48 bytes of data to be passed to the server. Note that the negotiated algorithms are not explicitly passed, as the algorithms given in Section ``Sending Supported Algorithm Lists'' fully determine the algorithms.

Note that the use of SHA-1 was hard-coded here. This is used to authenticate the key exchange, and using HASH here would lead to all sorts of potential problems in verifying the security of the protocol. Using a fixed hash short-circuits verification to the properties of the hash. Should the need ever arise, the only way to switch to another algorithm here is to define a new key exchange algorithm (which, in fact, is not very difficult).

The resulting data is encrypted with the smaller of host key and server key, and the result then with the larger of them. The resulting double-encrypted session key is then sent to the server for verification. Note that public-key encryption probably involves padding, depending on the algorithm.

```
vlint32   SSH_MSG_KEXRSA_SESSIONKEY
string    double-encrypted session key
```

Upon receiving this message, the server uses its private host and server keys to decrypt the session key. It computes a corresponding SHA hash, and compares the hash values. If the hash does not match, the server disconnects. Otherwise, encryption, compression, and integrity protection are taken into effect immediately after this message.

The server does not acknowledge this message in any way. The client may continue by sending further protocol requests using the negotiated encryption. If the server was not able to decrypt the session key, it won't be able to determine what the futher requests are or to respond to

them.

5.3. Diffie-Hellman Style Key Exchange

XXX To be defined later.

5.4. Deriving Encryption and Integrity Keys

As a result of the key exchange, the parties have a 256-bit shared secret.

Various keys are computed from this secret and from the cookies exchanged during algorithm negotiation. The cookies are used to make it impossible for either party to alone determine the keys.

Each key is computed as HASH of the concatenation of client's cookie, server's cookie, and 16 bytes of secret data. The secret data is different for each key, and is taken from the 32-byte shared secret as follows:

- Initial IV client to server: bytes 0-15
- Initial IV server to client: bytes 1-16
- Encryption key client to server: bytes 5-20
- Encryption key server to client: bytes 8-23
- Integrity key client to server: bytes 13-28
- Integrity key server to client: bytes 16-31

Each key is at least 16 bytes (128 bits). For some algorithms, only part of this amount is actually used. If a longer key is needed for some algorithm, the key is extended by computing HASH of the entire key so far, and appending the resulting bytes (as many as HASH outputs) to the key. This process is repeated until enough key material is available.

6. Client Host Authentication

Next, the client can (but is not required to) send its own host authentication message. If this message is not sent, the server will

consider the client unnamed for authentication purposes. It is expected that many servers will refuse to talk to clients that do not first authenticate themselves.

Note that this message passes the host name, and can be used to authenticate hosts on the other side of a firewall.

To authenticate itself, the client computes HASH of the concatenatenation of (in this order) the client host name, client's cookie, and server's cookie. It then signs this hash using its private host key (one that uses an algorithm which is supported by the server), and sends the following value to the server:

```
vlint32   SSH_MSG_HOSTAUTH
string    client host name
string    client public host key
string    signature of host name+client cookie+server cookie
```

The host key is stored in binary representation as described in Section ``Public Key Formats''.

The server will verify that the client host name is really associated with the given key. If the name-key association cannot be verified, the server may at its option either disconnect or ignore the client host authentication.

If the name-key association can be verified, the signature is checked. If it is incorrect, the connection is closed. Otherwise, the server updates its state to the fact that the client has been cryptographically authenticated.

The server may also attempt to validate whether the host name supplied by the client matches with a name obtained from network protocol headers or other sources (e.g., it may require the supplied name to match a DNS reverse mapped name, unless the reverse mapped name is a firewall host). However, such validation is not always possible, and is not required by this protocol. If the server finds a mismatch between the client-supplied name and the name obtained from the network protocols, the server may disconnect.

Some servers may refuse to continue the dialog with the client unless the client is able to authenticate itself cryptographically. Likewise, some servers may refuse to talk to certain clients, and may disconnect at this point. It is recommended that such disconnections use

SSH_MSG_DISCONNECT and explain the reason for disconnecting.

7. Service Request

After the various authentications, the client requests a service. The service is identified by a name, which must consists of alphanumeric characters, hyphens ('-'), and underscores ('_'). The name must not be longer than 64 characters.

```
vlint32   SSH_MSG_SERVICE_REQUEST
string    service name
```

Most server implementations will have a table of services that are supported, specifying what to do for each service.

If the server rejects the service request, it either disconnects or (preferably) sends a SSH_MSG_DISCONNECT message.

If the server supports the service (and permits the client to use it), it responds with

```
vlint32   SSH_MSG_SERVICE_ACCEPT
```

The client is permitted to send further packets without waiting for the this message; those packets will go the selected service if the server accepts the service request.

8. Data Exchange

Once a service has been selected, data is transmitted in each direction asynchronously. The data is packetized using the following format:

```
vlint32   SSH_MSG_DATA
string    data
```


For stream-based services data is passed directly to the application. Packet-based applications may have their own packet structure embedded within each data packet.

When a service closes its output (i.e., will not send more data), the following message is sent to the other side:

```
vlint32   SSH_MSG_EOF
```

When this message is received, the service will be notified of the end-of-file status on input. Output from the service is still accepted, and sent to the other side.

When the service on either side exits (i.e., will no longer accept input, and will not generate more output), the following message is sent after any other data sent by the application. After this message, the connection will be closed. An EOF is not necessarily sent before this message.

```
            vlint32   SSH_MSG_CLOSE
```

9. Key Re-Exchange

Either side may request re-exchange of keys at any time after the initial exchange (and outside other key exchanges). The re-exchange is not visible to the service, and will take place using the same algorithm that was used in the original key exchange.

Key re-exchange is started by sending a SSH_MSG_KEXINIT packet (described in Section ``Sending Supported Algorithm Lists''). When this message is received, a party must respond with its own SSH_MSG_KEXINIT message. Either party may initiate the re-exchange, but roles are not changed (i.e., the server remains the server, and the client remains the client).

Key re-exchange is performed under whatever encryption was in effect when the exchange was started. Encryption, compression, and MAC methods are changed when the key exchange is complete (as in the initial key exchange). Re-exchange is processed identically to the initial key exchange, except that it is not necessary to validate that the host key really belongs to the server. It is permissible to change the encryption, compression, and MAC algorithms during the re-exchange. All keys are recomputed after the exchange. Compression and encryption contexts are reset. The packet sequence number is not reset.

It is recommended that keys be changed after each gigabyte of transmitted data and after each hour of connection time, whichever comes sooner.

It is also possible to use the key re-exchange mechanism to switch to faster algorithms after authentication, or to avoid double processing for pre-encrypted or pre-authenticated data. 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.

10. Additional Messages

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

10.1. Disconnection Message

```
            vlint32   SSH_MSG_DISCONNECT
            vlint32   error code
            string    description
```

This message causes immediate termination of the connection. The description field gives the reason for disconnecting in human-readable form. The error code gives the reason in a machine-readable format, and can have the following values:

```
#define SSH_DISCONNECT_HOST_NOT_ALLOWED_TO_CONNECT      1
#define SSH_DISCONNECT_PROTOCOL_ERROR                   2
#define SSH_DISCONNECT_INCOMPATIBLE_HOST_KEY            3
#define SSH_DISCONNECT_INCOMPATIBLE_PUBLIC_KEY_ALG      4
#define SSH_DISCONNECT_INCOMPATIBLE_CRYPTO_ALG          5
#define SSH_DISCONNECT_INCOMPATIBLE_MAC_ALG             6
#define SSH_DISCONNECT_INCOMPATIBLE_COMPRESSION_ALG     7
#define SSH_DISCONNECT_INCOMPATIBLE_HASH_ALG            8
#define SSH_DISCONNECT_KEY_EXCHANGE_FAILED              9
#define SSH_DISCONNECT_HOST_AUTHENTICATION_FAILED       10
#define SSH_DISCONNECT_MAC_ERROR                        11
#define SSH_DISCONNECT_COMPRESSION_ERROR                12
#define SSH_DISCONNECT_SERVICE_NOT_AVAILABLE            13
#define SSH_DISCONNECT_PROTOCOL_VERSION_NOT_SUPPORTED   14
#define SSH_DISCONNECT_OPERATION_PROHIBITED             15
```

10.2. Ignored Data Message

```
            vlint32   SSH_MSG_IGNORE
            string    data
```

All implementations must understand (and ignore) this message. No implementation is required to ever send them.

10.3. Reserved Messages

An implementation must respond to all unrecognized messages with an SSH_MSG_UNIMPLEMENTED message. Later protocol versions may define other meanings for these message types.

```
            vlint32   SSH_MSG_UNIMPLEMENTED
```

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

Message numbers below 1000 are reserved for "official" extensions. Other extensions and experimental code should use message numbers above this.

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_KEXINIT               10
#define SSH_MSG_KEXRSA_HOSTKEY        11
#define SSH_MSG_KEXRSA_SESSIONKEY     12
#define SSH_MSG_HOSTAUTH              30
#define SSH_MSG_SERVICE_REQUEST       40
#define SSH_MSG_SERVICE_ACCEPT        41
#define SSH_MSG_DATA                  42
#define SSH_MSG_EOF                   43
#define SSH_MSG_CLOSE                 44
```

12. Public Keys and Public Key Infrastructure

This protocol is intentionally open on public key formats, as well as signature and encryption formats. There is currently no generally accepted public key infrastructure on the Internet, and there are several competing key formats, and more formats are likely to appear. It will probably take several years until the situation is resolved. In particular, it is not clear that X.509 would be the solution, although that is also a possibility.

There are several aspects to a public key type:

- Key format: how is the key coded, and how are certificates represented. The key blobs in this protocol may (but are not required to) contain certificates in addition to keys.

- Signature and/or encryption algorithms. Some algorithms may not support both encryption and decryption.

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

o Computation of unique key id.

```
#define SSH_PK_SIMPLE_RSA_PKCS    1
```

Note that the key type is negotiated at the beginning of the key exchange, and is not included in the key blob itself.

12.1. SSH_PK_SIMPLE_RSA_PKCS

This key type defines an RSA public key, with (mostly) PKCS compatible signature and encryption formats. It supports both signatures and encryption.

Public keys of this type are represented as follows:

byte[4]	0 (reserved)
uint32	number of bits in the modulus
uint16	number bits in the secret exponent
bytes[n]	exponent, MSB first, n = floor((bits+7)/8)
uint16	number of bits in the modulus
bytes[n]	modulus, MSB first, n = floor((bits+7)/8)

It is permissible that there be other data (e.g., certificates) following this; however, such data is not yet defined.

Note that private key formats are not defined here, and are implementation-specific.

The key id for this key type is of the format "SSH-PKCS-1-RSA-xxxxxxxx", where the "xxxxxxxx" stands for an SHA-1 hash of the data listed above, in hexadecimal (a total of 40 lowercase hex characters). Any data following the above (e.g., certificates) is not included in the key id calculation.

An encrypted message is formed as follows.

o The data to be encrypted is padded into a long integer of the same number of bits as the modulus as follows:

```
MSB              .  .  .                LSB

 0   2   RND(n bytes)   0   encrypted_data
```

The RND bytes represent non-zero random bytes.

- To encrypt, this integer is raised to the public exponent, modulo the modulus.

- The result is converted to a byte string of floor((bits+7)/8) bytes (where bits is the number of bits in the modulus), MSB first. This byte string (without any length or terminating characters) is the result of the encryption.

A signature is formed as follows.

- The data to be signed (typically a message digest, but not required to be such) is padded into a long integer of the same number of bits as the modulus as follows:

```
            MSB              .  .  .              LSB

             0   1   RND(n bytes)   0   signed_data
```

The RND bytes represent non-zero random bytes. Note that this differs from the PKCS standard, where 0xFF bytes are specified for padding.

- To sign, this integer is raised to the private exponent, modulo the modulus.

- The result is converted to a byte string of floor((bits+7)/8) bytes (where bits is the number of bits in the modulus), MSB first. This byte string (without any length or terminating characters) is the signature. Applications may add other data outside this value.