The SSH (Secure Shell) Remote Login Protocol

Status of This Memo

This document is an Internet-Draft. Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

To learn the current status of any Internet-Draft, please check the "id-abstracts.txt" listing contained in the Internet-Drafts Shadow Directories on ftp.is.co.za (Africa), nic.nordu.net (Europe), munnari.oz.au (Pacific Rim), ds.internic.net (US East Coast), or ftp.isi.edu (US West Coast).

The distribution of this memo is unlimited.

Introduction

SSH (Secure Shell) is a program to log into another computer over a network, to execute commands in a remote machine, and to move files from one machine to another. It provides strong authentication and secure communications over insecure networks. Its features include the following:

- Closes several security holes (e.g., IP, routing, and DNS spoofing). New authentication methods: .rhosts together with RSA [RSA] based host authentication, and pure RSA authentication.
- All communications are automatically and transparently encrypted. Encryption is also used to protect integrity.
- X11 connection forwarding provides secure X11 sessions.
- Arbitrary TCP/IP ports can be redirected over the encrypted channel in both directions.
Client RSA-authenticates the server machine in the beginning of every connection to prevent trojan horses (by routing or DNS spoofing) and man-in-the-middle attacks, and the server RSA-authenticates the client machine before accepting .rhosts or /etc/hosts.equiv authentication (to prevent DNS, routing, or IP spoofing).

An authentication agent, running in the user’s local workstation or laptop, can be used to hold the user’s RSA authentication keys.

The goal has been to make the software as easy to use as possible for ordinary users. The protocol has been designed to be as secure as possible while making it possible to create implementations that are easy to use and install. The sample implementation has a number of convenient features that are not described in this document as they are not relevant for the protocol.

Overview of the Protocol

The software consists of a server program running on a server machine, and a client program running on a client machine (plus a few auxiliary programs). The machines are connected by an insecure IP network (that can be monitored, tampered with, and spoofed by hostile parties).

A connection is always initiated by the client side. The server listens on a specific port waiting for connections. Many clients may connect to the same server machine.

The client and the server are connected via a TCP/IP socket that is used for bidirectional communication. Other types of transport can be used but are currently not defined.

When the client connects the server, the server accepts the connection and responds by sending back its version identification string. The client parses the server’s identification, and sends its own identification. The purpose of the identification strings is to validate that the connection was to the correct port, declare the protocol version number used, and to declare the software version used on each side (for debugging purposes). The identification strings are human-readable. If either side fails to understand or support the other side’s version, it closes the connection.

After the protocol identification phase, both sides switch to a packet based binary protocol. The server starts by sending its host key (every host has an RSA key used to authenticate the host), server
key (an RSA key regenerated every hour), and other information to the client. The client then generates a 256 bit session key, encrypts it using both RSA keys (see below for details), and sends the encrypted session key and selected cipher type to the server. Both sides then turn on encryption using the selected algorithm and key. The server sends an encrypted confirmation message to the client.

The client then authenticates itself using any of a number of authentication methods. The currently supported authentication methods are .rhosts or /etc/hosts.equiv authentication (disabled by default), the same with RSA-based host authentication, RSA authentication, and password authentication.

After successful authentication, the client makes a number of requests to prepare for the session. Typical requests include allocating a pseudo tty, starting X11 [X11] or TCP/IP port forwarding, starting authentication agent forwarding, and executing the shell or a command.

When a shell or command is executed, the connection enters interactive session mode. In this mode, data is passed in both directions, new forwarded connections may be opened, etc. The interactive session normally terminates when the server sends the exit status of the program to the client.

The protocol makes several reservations for future extensibility. First of all, the initial protocol identification messages include the protocol version number. Second, the first packet by both sides includes a protocol flags field, which can be used to agree on extensions in a compatible manner. Third, the authentication and session preparation phases work so that the client sends requests to the server, and the server responds with success or failure. If the client sends a request that the server does not support, the server simply returns failure for it. This permits compatible addition of new authentication methods and preparation operations. The interactive session phase, on the other hand, works asynchronously and does not permit the use of any extensions (because there is no easy and reliable way to signal rejection to the other side and problems would be hard to debug). Any compatible extensions to this phase must be agreed upon during any of the earlier phases.

The Binary Packet Protocol

After the protocol identification strings, both sides only send specially formatted packets. The packet layout is as follows:

- Packet length: 32 bit unsigned integer, coded as four 8-bit
bytes, msb first. Gives the length of the packet, not including the length field and padding. The maximum length of a packet (not including the length field and padding) is 262144 bytes.

- Padding: 1-8 bytes of random data (or zeroes if not encrypting). The amount of padding is \((8 - (\text{length} \mod 8))\) bytes (where \(\%\) stands for the modulo operator). The rationale for always having some random padding at the beginning of each packet is to make known plaintext attacks more difficult.

- Packet type: 8-bit unsigned byte. The value 255 is reserved for future extension.

- Data: binary data bytes, depending on the packet type. The number of data bytes is the "length" field minus 5.

- Check bytes: 32-bit crc, four 8-bit bytes, msb first. The crc is the Cyclic Redundancy Check, with the polynomial 0xedb88320, of the Padding, Packet type, and Data fields. The crc is computed before any encryption.

The packet, except for the length field, may be encrypted using any of a number of algorithms. The length of the encrypted part (Padding + Type + Data + Check) is always a multiple of 8 bytes. Typically the cipher is used in a chained mode, with all packets chained together as if it was a single data stream (the length field is never included in the encryption process). Details of encryption are described below.

When the session starts, encryption is turned off. Encryption is enabled after the client has sent the session key. The encryption algorithm to use is selected by the client.

Packet Compression

If compression is supported (it is an optional feature, see SSH_CMSG_REQUEST_COMPRESSION below), the packet type and data fields of the packet are compressed using the gzip deflate algorithm [GZIP]. If compression is in effect, the packet length field indicates the length of the compressed data, plus 4 for the crc. The amount of padding is computed from the compressed data, so that the amount of data to be encrypted becomes a multiple of 8 bytes.

When compressing, the packets (type + data portions) in each direction are compressed as if they formed a continuous data stream, with only the current compression block flushed between packets. This corresponds to the GNU ZLIB library Z_PARTIAL_FLUSH option. The
compression dictionary is not flushed between packets. The two
directions are compressed independently of each other.

Packet Encryption

The protocol supports several encryption methods. During session
initialization, the server sends a bitmask of all encryption methods
that it supports, and the client selects one of these methods. The
client also generates a 256-bit random session key (32 8-bit bytes)
and sends it to the server.

The encryption methods supported by the current implementation, and
their codes are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SSH_CIPHER_NONE No encryption</td>
</tr>
<tr>
<td>1</td>
<td>SSH_CIPHER_IDEA IDEA in CFB mode</td>
</tr>
<tr>
<td>2</td>
<td>SSH_CIPHER_DES DES in CBC mode</td>
</tr>
<tr>
<td>3</td>
<td>SSH_CIPHER_3DES Triple-DES in CBC mode</td>
</tr>
<tr>
<td>4</td>
<td>SSH_CIPHER_TSS An experimental stream cipher</td>
</tr>
<tr>
<td>5</td>
<td>SSH_CIPHER_RC4 RC4</td>
</tr>
</tbody>
</table>

All implementations are required to support SSH_CIPHER_DES and
SSH_CIPHER_3DES. Supporting SSH_CIPHER_IDEA, SSH_CIPHER_RC4, and
SSH_CIPHER_NONE is recommended. Support for SSH_CIPHER_TSS is
optional (and it is not described in this document). Other ciphers
may be added at a later time; support for them is optional.

For encryption, the encrypted portion of the packet is considered a
linear byte stream. The length of the stream is always a multiple of
8. The encrypted portions of consecutive packets (in the same direc-
tion) are encrypted as if they were a continuous buffer (that is, any
initialization vectors are passed from the previous packet to the
next packet). Data in each direction is encrypted independently.

**SSH_CIPHER_DES**
The key is taken from the first 8 bytes of the session key. The
least significant bit of each byte is ignored. This results in
56 bits of key data. DES [DES] is used in CBC mode. The iv
(initialization vector) is initialized to all zeroes.

**SSH_CIPHER_3DES**
The variant of triple-DES used here works as follows: there are
three independent DES-CBC ciphers, with independent initializa-
tion vectors. The data (the whole encrypted data stream) is
first encrypted with the first cipher, then decrypted with the
second cipher, and finally encrypted with the third cipher. All
these operations are performed in CBC mode.

The key for the first cipher is taken from the first 8 bytes of the session key; the key for the next cipher from the next 8 bytes, and the key for the third cipher from the following 8 bytes. All three initialization vectors are initialized to zero.

(Note: the variant of 3DES used here differs from some other descriptions.)

**SSH_CIPHER_IDEA**

The key is taken from the first 16 bytes of the session key. IDEA [IDEA] is used in CFB mode. The initialization vector is initialized to all zeroes.

**SSH_CIPHER_TSS**

All 32 bytes of the session key are used as the key.

There is no reference available for the TSS algorithm; it is currently only documented in the sample implementation source code. The security of this cipher is unknown (but it is quite fast). The cipher is basically a stream cipher that uses MD5 as a random number generator and takes feedback from the data.

**SSH_CIPHER_RC4**

The first 16 bytes of the session key are used as the key for the server to client direction. The remaining 16 bytes are used as the key for the client to server direction. This gives independent 128-bit keys for each direction.

This algorithm is the alleged RC4 cipher posted to the Usenet in 1995. It is widely believed to be equivalent with the original RSADSI RC4 cipher. This is a very fast algorithm.

Data Type Encodings

The Data field of each packet contains data encoded as described in this section. There may be several data items; each item is coded as described here, and their representations are concatenated together (without any alignment or padding).

Each data type is stored as follows:

8-bit byte

The byte is stored directly as a single byte.
32-bit unsigned integer
   Stored in 4 bytes, msb first.

Arbitrary length binary string
   First 4 bytes are the length of the string, msb first (not
   including the length itself). The following "length" bytes are
   the string value. There are no terminating null characters.

Multiple-precision integer
   First 2 bytes are the number of bits in the integer, msb first
   (for example, the value 0x00012345 would have 17 bits). The
   value zero has zero bits. It is permissible that the number of
   bits be larger than the real number of bits.

   The number of bits is followed by (bits + 7) / 8 bytes of binary
   data, msb first, giving the value of the integer.

TCP/IP Port Number and Other Options

   The server listens for connections on TCP/IP port 22.

   The client may connect the server from any port. However, if the
   client wishes to use any form of .rhosts or /etc/hosts.equiv authen-
   tication, it must connect from a privileged port (less than 1024).

   For the IP Type of Service field [RFC0791], it is recommended that
   interactive sessions (those having a user terminal or forwarding X11
   connections) use the IPTOS_LOWDELAY, and non-interactive connections
   use IPTOS_THROUGHPUT.

   It is recommended that keepalives are used, because otherwise pro-
   grams on the server may never notice if the other end of the connec-
   tion is rebooted.

Protocol Version Identification

   After the socket is opened, the server sends an identification
   string, which is of the form "SSH-<protocolmajor>.<protocolminor>-
   <version>\n", where <protocolmajor> and <protocolminor> are integers
   and specify the protocol version number (not software distribution
   version). <version> is server side software version string (max 40
   characters); it is not interpreted by the remote side but may be use-
   ful for debugging.

   The client parses the server’s string, and sends a corresponding
   string with its own information in response. If the server has lower
version number, and the client contains special code to emulate it, the client responds with the lower number; otherwise it responds with its own number. The server then compares the version number the client sent with its own, and determines whether they can work together. The server either disconnects, or sends the first packet using the binary packet protocol and both sides start working according to the lower of the protocol versions.

By convention, changes which keep the protocol compatible with previous versions keep the same major protocol version; changes that are not compatible increment the major version (which will hopefully never happen). The version described in this document is 1.3.

The client will

Key Exchange and Server Host Authentication

The first message sent by the server using the packet protocol is SSH_SMSG_PUBLIC_KEY. It declares the server’s host key, server public key, supported ciphers, supported authentication methods, and flags for protocol extensions. It also contains a 64-bit random number (cookie) that must be returned in the client’s reply (to make IP spoofing more difficult). No encryption is used for this message.

Both sides compute a session id as follows. The modulus of the server key is interpreted as a byte string (without explicit length field, with minimum length able to hold the whole value), most significant byte first. This string is concatenated with the server host key interpreted the same way. Additionally, the cookie is concatenated with this. Both sides compute MD5 of the resulting string. The resulting 16 bytes (128 bits) are stored by both parties and are called the session id.

The client responds with a SSH_CMSG_SESSION_KEY message, which contains the selected cipher type, a copy of the 64-bit cookie sent by the server, client’s protocol flags, and a session key encrypted with both the server’s host key and server key. No encryption is used for this message.

The session key is 32 8-bit bytes (a total of 256 random bits generated by the client). The client first xors the 16 bytes of the session id with the first 16 bytes of the session key. The resulting string is then encrypted using the smaller key (one with smaller modulus), and the result is then encrypted using the other key. The number of bits in the public modulus of the two keys must differ by at least 128 bits.

At each encryption step, a multiple-precision integer is constructed
from the data to be encrypted as follows (the integer is here inter-
preted as a sequence of bytes, msb first; the number of bytes is the
number of bytes needed to represent the modulus).

The most significant byte (which is only partial as the value must be
less than the public modulus, which is never a power of two) is zero.

The next byte contains the value 2 (which stands for public-key
encrypted data in the PKCS standard [PKCS#1]). Then, there are non-
zero random bytes to fill any unused space, a zero byte, and the data
to be encrypted in the least significant bytes, the last byte of the
data in the least significant byte.

This algorithm is used twice. First, it is used to encrypt the 32
random bytes generated by the client to be used as the session key
(xored by the session id). This value is converted to an integer as
described above, and encrypted with RSA using the key with the
smaller modulus. The resulting integer is converted to a byte
stream, msb first. This byte stream is padded and encrypted identi-
cally using the key with the larger modulus.

After the client has sent the session key, it starts to use the
selected algorithm and key for decrypting any received packets, and
for encrypting any sent packets. Separate ciphers are used for dif-
ferent directions (that is, both directions have separate initializa-
tion vectors or other state for the ciphers).

When the server has received the session key message, and has turned
on encryption, it sends a SSH_SMSG_SUCCESS message to the client.

The recommended size of the host key is 1024 bits, and 768 bits for
the server key. The minimum size is 512 bits for the smaller key.

Declaring the User Name

The client then sends a SSH_CMSG_USER message to the server. This
message specifies the user name to log in as.

The server validates that such a user exists, checks whether authen-
tication is needed, and responds with either SSH_SMSG_SUCCESS or
SSH_SMSG_FAILURE. SSH_SMSG_SUCCESS indicates that no authentication
is needed for this user (no password), and authentication phase has
now been completed. SSH_SMSG_FAILURE indicates that authentication
is needed (or the user does not exist).

If the user does not exist, it is recommended that this returns
failure, but the server keeps reading messages from the client, and
responds to any messages (except SSH_MSG_DISCONNECT, SSH_MSG_IGNORE,
and SSH_MSG_DEBUG) with SSH_SMSG_FAILURE. This way the client cannot
be certain whether the user exists.

Authentication Phase

Provided the server didn’t immediately accept the login, an authenti-
cation exchange begins. The client sends messages to the server
requesting different types of authentication in arbitrary order as
many times as desired (however, the server may close the connection
after a timeout). The server always responds with SSH_SMSG_SUCCESS
if it has accepted the authentication, and with SSH_SMSG_FAILURE if
it has denied authentication with the requested method or it does not
recognize the message. Some authentication methods cause an exchange
of further messages before the final result is sent. The authentica-
tion phase ends when the server responds with success.

The recommended value for the authentication timeout (timeout before
disconnecting if no successful authentication has been made) is 5
minutes.

The following authentication methods are currently supported:

- **SSH_AUTH_RHOSTS** 1 .rhosts or /etc/hosts.equiv
- **SSH_AUTH_RSA** 2 pure RSA authentication
- **SSH_AUTH_PASSWORD** 3 password authentication
- **SSH_AUTH_RHOSTS_RSA** 4 .rhosts with RSA host authentication

**SSH_AUTH_RHOSTS**

This is the authentication method used by rlogin and rsh
[RFC1282].

The client sends SSH_CMSG_AUTH_RHOSTS with the client-side user
name as an argument.

The server checks whether to permit authentication. On UNIX
systems, this is usually done by checking /etc/hosts.equiv, and
.rhosts in the user’s home directory. The connection must come
from a privileged port.

It is recommended that the server checks that there are no IP
options (such as source routing) specified for the socket before
accepting this type of authentication. The client host name
should be reverse-mapped and then forward mapped to ensure that
it has the proper IP-address.
This authentication method trusts the remote host (root on the remote host can pretend to be any other user on that host), the name services, and partially the network: anyone who can see packets coming out from the server machine can do IP-spoofing and pretend to be any machine; however, the protocol prevents blind IP-spoofing (which used to be possible with rlogin).

Many sites probably want to disable this authentication method because of the fundamental insecurity of conventional .rhosts or /etc/hosts.equiv authentication when faced with spoofing. It is recommended that this method not be supported by the server by default.

**SSH_AUTH_RHOSTS_RSA**

In addition to conventional .rhosts and hosts.equiv authentication, this method additionally requires that the client host be authenticated using RSA.

The client sends SSH_CMSG_AUTH_RHOSTS_RSA specifying the client-side user name, and the public host key of the client host.

The server first checks if normal .rhosts or /etc/hosts.equiv authentication would be accepted, and if not, responds with SSH_SMSG_FAILURE. Otherwise, it checks whether it knows the host key for the client machine (using the same name for the host that was used for checking the .rhosts and /etc/hosts.equiv files). If it does not know the RSA key for the client, access is denied and SSH_SMSG_FAILURE is sent.

If the server knows the host key of the client machine, it verifies that the given host key matches that known for the client. If not, access is denied and SSH_SMSG_FAILURE is sent.

The server then sends a SSH_SMSG_AUTH_RSA_CHALLENGE message containing an encrypted challenge for the client. The challenge is 32 8-bit random bytes (256 bits). When encrypted, the highest (partial) byte is left as zero, the next byte contains the value 2, the following are non-zero random bytes, followed by a zero byte, and the challenge put in the remaining bytes. This is then encrypted using RSA with the client host’s public key. (The padding and encryption algorithm is the same as that used for the session key.)

The client decrypts the challenge using its private host key, concatenates this with the session id, and computes an MD5 checksum of the resulting 48 bytes. The MD5 output is returned
as 16 bytes in a SSH_CMSG_AUTH_RSA_RESPONSE message. (MD5 is used to deter chosen plaintext attacks against RSA; the session id binds it to a specific session).

The server verifies that the MD5 of the decrypted challenge returned by the client matches that of the original value, and sends SSH_SMSG_SUCCESS if so. Otherwise it sends SSH_SMSG_FAILURE and refuses the authentication attempt.

This authentication method trusts the client side machine in that root on that machine can pretend to be any user on that machine. Additionally, it trusts the client host key. The name and/or IP address of the client host is only used to select the public host key. The same host name is used when scanning .rhosts or /etc/hosts.equiv and when selecting the host key. It would in principle be possible to eliminate the host name entirely and substitute it directly by the host key. IP and/or DNS [RFC1034] spoofing can only be used to pretend to be a host for which the attacker has the private host key.

SSH_AUTH_RSA

The idea behind RSA authentication is that the server recognizes the public key offered by the client, generates a random challenge, and encrypts the challenge with the public key. The client must then prove that it has the corresponding private key by decrypting the challenge.

The client sends SSH_CMSG_AUTH_RSA with public key modulus (n) as an argument.

The server may respond immediately with SSH_SMSG_FAILURE if it does not permit authentication with this key. Otherwise it generates a challenge, encrypts it using the user’s public key (stored on the server and identified using the modulus), and sends SSH_SMSG_AUTH_RSA_CHALLENGE with the challenge (mp-int) as an argument.

The challenge is 32 8-bit random bytes (256 bits). When encrypted, the highest (partial) byte is left as zero, the next byte contains the value 2, the following are non-zero random bytes, followed by a zero byte, and the challenge put in the remaining bytes. This is then encrypted with the public key. (The padding and encryption algorithm is the same as that used for the session key.)

The client decrypts the challenge using its private key, concatenates it with the session id, and computes an MD5 checksum.
of the resulting 48 bytes. The MD5 output is returned as 16 bytes in a SSH_CMSG_AUTH_RSA_RESPONSE message. (Note that the MD5 is necessary to avoid chosen plaintext attacks against RSA; the session id binds it to a specific session.)

The server verifies that the MD5 of the decrypted challenge returned by the client matches that of the original value, and sends SSH_SMSG_SUCCESS if so. Otherwise it sends SSH_SMSG_FAILURE and refuses the authentication attempt.

This authentication method does not trust the remote host, the network, name services, or anything else. Authentication is based solely on the possession of the private identification keys. Anyone in possession of the private keys can log in, but nobody else.

The server may have additional requirements for a successful authentication. For example, to limit damage due to a compromised RSA key, a server might restrict access to a limited set of hosts.

**SSH_AUTH_PASSWORD**

The client sends a SSH_CMSG_AUTH_PASSWORD message with the plain text password. (Note that even though the password is plain text inside the message, it is normally encrypted by the packet mechanism.)

The server verifies the password, and sends SSH_SMSG_SUCCESS if authentication was accepted and SSH_SMSG_FAILURE otherwise.

Note that the password is read from the user by the client; the user never interacts with a login program.

This authentication method does not trust the remote host, the network, name services or anything else. Authentication is based solely on the possession of the password. Anyone in possession of the password can log in, but nobody else.

**Preparatory Operations**

After successful authentication, the server waits for a request from the client, processes the request, and responds with SSH_SMSG_SUCCESS whenever a request has been successfully processed. If it receives a message that it does not recognize or it fails to honor a request, it returns SSH_SMSG_FAILURE. It is expected that new message types might be added to this phase in future.
The following messages are currently defined for this phase.

**SSH_CMSG_REQUEST_COMPRESSION**
Requests that compression be enabled for this session. A gzip-compatible compression level (1-9) is passed as an argument.

**SSH_CMSG_REQUEST_PTY**
Requests that a pseudo terminal device be allocated for this session. The user terminal type and terminal modes are supplied as arguments.

**SSH_CMSG_X11_REQUEST_FORWARDING**
Requests forwarding of X11 connections from the remote machine to the local machine over the secure channel. Causes an internet-domain socket to be allocated and the DISPLAY variable to be set on the server. X11 authentication data is automatically passed to the server, and the client may implement spoofing of authentication data for added security. The authentication data is passed as arguments.

**SSH_CMSG_PORT_FORWARD_REQUEST**
Requests forwarding of a TCP/IP port on the server host over the secure channel. What happens is that whenever a connection is made to the port on the server, a connection will be made from the client end to the specified host/port. Any user can forward unprivileged ports; only the root can forward privileged ports (as determined by authentication done earlier).

**SSH_CMSG_AGENT_REQUEST_FORWARDING**
Requests forwarding of the connection to the authentication agent.

**SSH_CMSG_EXEC_SHELL**
Starts a shell (command interpreter) for the user, and moves into interactive session mode.

**SSH_CMSG_EXEC_CMD**
Executes the given command (actually "<shell> -c <command>" or equivalent) for the user, and moves into interactive session mode.

### Interactive Session and Exchange of Data

During the interactive session, any data written by the shell or command running on the server machine is forwarded to stdin or stderr on the client machine, and any input available from stdin on the client machine is forwarded to the program on the server machine.
All exchange is asynchronous; either side can send at any time, and there are no acknowledgements (TCP/IP already provides reliable transport, and the packet protocol protects against tampering or IP spoofing).

When the client receives EOF from its standard input, it will send SSH_CMSG_EOF; however, this in no way terminates the exchange. The exchange terminates and interactive mode is left when the server sends SSH_SMSG_EXITSTATUS to indicate that the client program has terminated. Alternatively, either side may disconnect at any time by sending SSH_MSG_DISCONNECT or closing the connection.

The server may send any of the following messages:

**SSH_SMSG_STDOUT_DATA**
Data written to stdout by the program running on the server. The data is passed as a string argument. The client writes this data to stdout.

**SSH_SMSG_STDERR_DATA**
Data written to stderr by the program running on the server. The data is passed as a string argument. The client writes this data to stderr. (Note that if the program is running on a tty, it is not possible to separate stdout and stderr data, and all data will be sent as stdout data.)

**SSH_SMSG_EXITSTATUS**
Indicates that the shell or command has exited. Exit status is passed as an integer argument. This message causes termination of the interactive session.

**SSH_SMSG_AGENT_OPEN**
Indicates that someone on the server side is requesting a connection to the authentication agent. The server-side channel number is passed as an argument. The client must respond with either SSH_CHANNEL_OPEN_CONFIRMATION or SSH_CHANNEL_OPEN_FAILURE.

**SSH_SMSG_X11_OPEN**
Indicates that a connection has been made to the X11 socket on the server side and should be forwarded to the real X server. An integer argument indicates the channel number allocated for this connection on the server side. The client should send back either SSH_MSG_CHANNEL_OPEN_CONFIRMATION or SSH_MSG_CHANNEL_OPEN_FAILURE with the same server side channel number.

**SSH_MSG_PORT_OPEN**
Indicates that a connection has been made to a port on the server side for which forwarding has been requested. Arguments are server side channel number, host name to connect to, and port to connect to. The client should send back either SSH_MSG_CHANNEL_OPEN_CONFIRMATION or SSH_MSG_CHANNEL_OPEN_FAILURE with the same server side channel number.

SSH_MSG_CHANNEL_OPEN_CONFIRMATION
This is sent by the server to indicate that it has opened a connection as requested in a previous message. The first argument indicates the client side channel number, and the second argument is the channel number that the server has allocated for this connection.

SSH_MSG_CHANNEL_OPEN_FAILURE
This is sent by the server to indicate that it failed to open a connection as requested in a previous message. The client-side channel number is passed as an argument. The client will close the descriptor associated with the channel and free the channel.

SSH_MSG_CHANNEL_DATA
This packet contains data for a channel from the server. The first argument is the client-side channel number, and the second argument (a string) is the data.

SSH_MSG_CHANNEL_CLOSE
This is sent by the server to indicate that whoever was in the other end of the channel has closed it. The argument is the client side channel number. The client will let all buffered data in the channel to drain, and when ready, will close the socket, free the channel, and send the server a SSH_MSG_CHANNEL_CLOSE_CONFIRMATION message for the channel.

SSH_MSG_CHANNEL_CLOSE_CONFIRMATION
This is sent by the server to indicate that a channel previously closed by the client has now been closed on the server side as well. The argument indicates the client channel number. The client frees the channel.

The client may send any of the following messages:

SSH_CMSG_STDIN_DATA
This is data to be sent as input to the program running on the server. The data is passed as a string.

SSH_CMSG.EOF
Indicates that the client has encountered EOF while reading
standard input. The server will allow any buffered input data to drain, and will then close the input to the program.

**SSH_CMSG_WINDOW_SIZE**
Indicates that window size on the client has been changed. The server updates the window size of the tty and causes SIGWINCH to be sent to the program. The new window size is passed as four integer arguments: row, col, xpixel, ypixel.

**SSH_MSG_PORT_OPEN**
Indicates that a connection has been made to a port on the client side for which forwarding has been requested. Arguments are client side channel number, host name to connect to, and port to connect to. The server should send back either SSH_MSG_CHANNEL_OPEN_CONFIRMATION or SSH_MSG_CHANNEL_OPEN_FAILURE with the same client side channel number.

**SSH_MSG_CHANNEL_OPEN_CONFIRMATION**
This is sent by the client to indicate that it has opened a connection as requested in a previous message. The first argument indicates the server side channel number, and the second argument is the channel number that the client has allocated for this connection.

**SSH_MSG_CHANNEL_OPEN_FAILURE**
This is sent by the client to indicate that it failed to open a connection as requested in a previous message. The server side channel number is passed as an argument. The server will close the descriptor associated with the channel and free the channel.

**SSH_MSG_CHANNEL_DATA**
This packet contains data for a channel from the client. The first argument is the server side channel number, and the second argument (a string) is the data.

**SSH_MSG_CHANNEL_CLOSE**
This is sent by the client to indicate that whoever was in the other end of the channel has closed it. The argument is the server channel number. The server will allow buffered data to drain, and when ready, will close the socket, free the channel, and send the client a SSH_MSG_CHANNEL_CLOSE_CONFIRMATION message for the channel.

**SSH_MSG_CHANNEL_CLOSE_CONFIRMATION**
This is send by the client to indicate that a channel previously closed by the server has now been closed on the client side as well. The argument indicates the server channel number. The
server frees the channel.

Any unsupported messages during interactive mode cause the connection to be terminated with SSH_MSG_DISCONNECT and an error message. Compatible protocol upgrades should agree about any extensions during the preparation phase or earlier.

**Termination of the Connection**

Normal termination of the connection is always initiated by the server by sending SSH_SMSG_EXITSTATUS after the program has exited. The client responds to this message by sending SSH_CMSG_EXIT_CONFIRMATION and closes the socket; the server then closes the socket. There are two purposes for the confirmation: some systems may lose previously sent data when the socket is closed, and closing the client side first causes any TCP/IP TIME_WAIT [RFC0793] waits to occur on the client side, not consuming server resources.

If the program terminates due to a signal, the server will send SSH_MSG_DISCONNECT with an appropriate message. If the connection is closed, all file descriptors to the program will be closed and the server will exit. If the program runs on a tty, the kernel sends it the SIGHUP signal when the pty master side is closed.

**Protocol Flags**

Both the server and the client pass 32 bits of protocol flags to the other side. The flags are intended for compatible protocol extension; the server first announces which added capabilities it supports, and the client then sends the capabilities that it supports.

The following flags are currently defined (the values are bit masks):

1. **SSH_PROTOFLAG_SCREEN_NUMBER**
   - This flag can only be sent by the client. It indicates that the X11 forwarding requests it sends will include the screen number.

2. **SSH_PROTOFLAG_HOST_IN_FWD_OPEN**
   - If both sides specify this flag, SSH_SMSG_X11_OPEN and SSH_MSG_PORT_OPEN messages will contain an additional field containing a description of the host at the other end of the connection.

**Detailed Description of Packet Types and Formats**

The supported packet types and the corresponding message numbers are given in the following table. Messages with _MSG_ in their name may
be sent by either side. Messages with _CMSG_ are only sent by the client, and messages with _SMSG_ only by the server.

A packet may contain additional data after the arguments specified below. Any such data should be ignored by the receiver. However, it is recommended that no such data be stored without good reason. (This helps build compatible extensions.)

0 SSH_MSG_NONE  
This code is reserved. This message type is never sent.

1 SSH_MSG_DISCONNECT  

string Cause of disconnection

This message may be sent by either party at any time. It causes the immediate disconnection of the connection. The message is intended to be displayed to a human, and describes the reason for disconnection.

2 SSH_SMSG_PUBLIC_KEY

8 bytes anti_spoofing_cookie
32-bit int server_key_bits
mp-int server_key_public_exponent
mp-int server_key_public_modulus
32-bit int host_key_bits
mp-int host_key_public_exponent
mp-int host_key_public_modulus
32-bit int protocol_flags
32-bit int supported_ciphers_mask
32-bit int supported_authentications_mask

Sent as the first message by the server. This message gives the server's host key, server key, protocol flags (intended for compatible protocol extension), supported_ciphers_mask (which is the bitwise or of (1 << cipher_number), where << is the left shift operator, for all supported ciphers), and supported_authentications_mask (which is the bitwise or of (1 << authentication_type) for all supported authentication types). The anti_spoofing_cookie is 64 random bytes, and must be sent back verbatim by the client in its reply. It is used to make IP-spoofing more difficult (encryption and host keys are the real defense against spoofing).

3 SSH_CMSG_SESSION_KEY

1 byte cipher_type (must be one of the supported values)
8 bytes    anti_spoofing_cookie (must match data sent by the server)
mp-int     double-encrypted session key
32-bit int  protocol_flags

Sent by the client as the first message in the session. Selects
the cipher to use, and sends the encrypted session key to the
server. The anti_spoofing_cookie must be the same bytes that
were sent by the server. Protocol_flags is intended for nego-
tiating compatible protocol extensions.

4 SSH_CMSG_USER

string   user login name on server

Sent by the client to begin authentication. Specifies the user
name on the server to log in as. The server responds with
SSH_SMSG_SUCCESS if no authentication is needed for this user,
or SSH_SMSG_FAILURE if authentication is needed (or the user
does not exist). [Note to the implementator: the user name is
of arbitrary size. The implementation must be careful not to
overflow internal buffers.]

5 SSH_CMSG_AUTH_RHOSTS

string   client-side user name

Requests authentication using /etc/hosts.equiv and .rhosts (or
equivalent mechanisms). This authentication method is normally
disabled in the server because it is not secure (but this is the
method used by rsh and rlogin). The server responds with
SSH_SMSG_SUCCESS if authentication was successful, and
SSH_SMSG_FAILURE if access was not granted. The server should
check that the client side port number is less than 1024 (a
privileged port), and immediately reject authentication if it is
not. Supporting this authentication method is optional. This
method should normally not be enabled in the server because it
is not safe. (However, not enabling this only helps if rlogind
and rshd are disabled.)

6 SSH_CMSG_AUTH_RSA

mp-int     identity_public_modulus

Requests authentication using pure RSA authentication. The
server checks if the given key is permitted to log in, and if
so, responds with SSH_SMSG_AUTH_RSA_CHALLENGE. Otherwise, it
responds with SSH_SMSG_FAILURE. The client often tries several
different keys in sequence until one supported by the server is
found. Authentication is accepted if the client gives the correct response to the challenge. The server is free to add other criteria for authentication, such as a requirement that the connection must come from a certain host. Such additions are not visible at the protocol level. Supporting this authentication method is optional but recommended.

7 SSH_SMSG_AUTH_RSA_CHALLENGE

mp-int encrypted challenge

Presents an RSA authentication challenge to the client. The challenge is a 256-bit random value encrypted as described elsewhere in this document. The client must decrypt the challenge using the RSA private key, compute MD5 of the challenge plus session id, and send back the resulting 16 bytes using SSH_CMSG_AUTH_RSA_RESPONSE.

8 SSH_CMSG_AUTH_RSA_RESPONSE

16 bytes MD5 of decrypted challenge

This message is sent by the client in response to an RSA challenge. The MD5 checksum is returned instead of the decrypted challenge to deter known-plaintext attacks against the RSA key. The server responds to this message with either SSH_SMSG_SUCCESS or SSH_SMSG_FAILURE.

9 SSH_CMSG_AUTH_PASSWORD

string plain text password

Requests password authentication using the given password. Note that even though the password is plain text inside the packet, the whole packet is normally encrypted by the packet layer. It would not be possible for the client to perform password encryption/hashing, because it cannot know which kind of encryption/hashing, if any, the server uses. The server responds to this message with either SSH_SMSG_SUCCESS or SSH_SMSG_FAILURE.

10 SSH_CMSG_REQUEST_PTY

string TERM environment variable value (e.g. vt100)
32-bit int terminal height, rows (e.g., 24)
32-bit int terminal width, columns (e.g., 80)
32-bit int terminal width, pixels (0 if no graphics) (e.g., 480)
32-bit int terminal height, pixels (0 if no graphics) (e.g., 640)
Requests a pseudo-terminal to be allocated for this command. This message can be used regardless of whether the session will later execute the shell or a command. If a pty has been requested with this message, the shell or command will run on a pty. Otherwise it will communicate with the server using pipes, sockets or some other similar mechanism.

The terminal type gives the type of the user’s terminal. In the UNIX environment it is passed to the shell or command in the TERM environment variable.

The width and height values give the initial size of the user’s terminal or window. All values can be zero if not supported by the operating system. The server will pass these values to the kernel if supported.

Terminal modes are encoded into a byte stream in a portable format. The exact format is described later in this document.

The server responds to the request with either SSH_SMSG_SUCCESS or SSH_SMSG_FAILURE. If the server does not have the concept of pseudo terminals, it should return success if it is possible to execute a shell or a command so that it looks to the client as if it was running on a pseudo terminal.

11 SSH_CMSG_WINDOW_SIZE

32-bit int  terminal height, rows
32-bit int  terminal width, columns
32-bit int  terminal width, pixels
32-bit int  terminal height, pixels

This message can only be sent by the client during the interactive session. This indicates that the size of the user’s window has changed, and provides the new size. The server will update the kernel’s notion of the window size, and a SIGWINCH signal or equivalent will be sent to the shell or command (if supported by the operating system).

12 SSH_CMSG_EXEC_SHELL

(no arguments)

Starts a shell (command interpreter), and enters interactive session mode.
13 SSH_CMSG_EXEC_CMD

  string   command to execute

  Starts executing the given command, and enters interactive ses-
  sion mode. On UNIX, the command is run as "<shell> -c <com-
  mand>", where <shell> is the user’s login shell.

14 SSH_SMSG_SUCCESS

  (no arguments)

  This message is sent by the server in response to the session
  key, a successful authentication request, and a successfully
  completed preparatory operation.

15 SSH_SMSG_FAILURE

  (no arguments)

  This message is sent by the server in response to a failed
  authentication operation to indicate that the user has not yet
  been successfully authenticated, and in response to a failed
  preparatory operation. This is also sent in response to an
  authentication or preparatory operation request that is not
  recognized or supported.

16 SSH_CMSG_STDIN_DATA

  string   data

  Delivers data from the client to be supplied as input to the
  shell or program running on the server side. This message can
  only be used in the interactive session mode. No acknowledge-
  ment is sent for this message.

17 SSH_SMSG_STDOUT_DATA

  string   data

  Delivers data from the server that was read from the standard
  output of the shell or program running on the server side. This
  message can only be used in the interactive session mode. No
  acknowledgement is sent for this message.

18 SSH_SMSG_STDERR_DATA

  string   data
Delivers data from the server that was read from the standard error of the shell or program running on the server side. This message can only be used in the interactive session mode. No acknowledgement is sent for this message.

19 SSH_CMSG_EOF

(no arguments)

This message is sent by the client to indicate that EOF has been reached on the input. Upon receiving this message, and after all buffered input data has been sent to the shell or program, the server will close the input file descriptor to the program. This message can only be used in the interactive session mode. No acknowledgement is sent for this message.

20 SSH_SMSG_EXITSTATUS

32-bit int exit status of the command

Returns the exit status of the shell or program after it has exited. The client should respond with SSH_CMSG_EXIT_CONFIRMATION when it has received this message. This will be the last message sent by the server. If the program being executed dies with a signal instead of exiting normally, the server should terminate the session with SSH_MSG_DISCONNECT (which can be used to pass a human-readable string indicating that the program died due to a signal) instead of using this message.

21 SSH_MSG_CHANNEL_OPEN_CONFIRMATION

32-bit int remote_channel
32-bit int local_channel

This is sent in response to any channel open request if the channel has been successfully opened. Remote_channel is the channel number received in the initial open request; local_channel is the channel number the side sending this message has allocated for the channel. Data can be transmitted on the channel after this message.

22 SSH_MSG_CHANNEL_OPEN_FAILURE

32-bit int remote_channel

This message indicates that an earlier channel open request by the other side has failed or has been denied. Remote_channel is
the channel number given in the original request.

23 SSH_MSG_CHANNEL_DATA

32-bit int   remote_channel
string       data

Data is transmitted in a channel in these messages. A channel is bidirectional, and both sides can send these messages. There is no acknowledgement for these messages. It is possible that either side receives these messages after it has sent SSH_MSG_CHANNEL_CLOSE for the channel. These messages cannot be received after the party has sent or received SSH_MSG_CHANNEL_CLOSE_CONFIRMATION.

24 SSH_MSG_CHANNEL_CLOSE

32-bit int   remote_channel

When a channel is closed at one end of the connection, that side sends this message. Upon receiving this message, the channel should be closed. When this message is received, if the channel is already closed (the receiving side has sent this message for the same channel earlier), the channel is freed and no further action is taken; otherwise the channel is freed and SSH_MSG_CHANNEL_CLOSE_CONFIRMATION is sent in response. (It is possible that the channel is closed simultaneously at both ends.)

25 SSH_MSG_CHANNEL_CLOSE_CONFIRMATION

32-bit int   remote_channel

This message is sent in response to SSH_MSG_CHANNEL_CLOSE unless the channel was already closed. When this message is sent or received, the channel is freed.

26 (OBSOLETED; was unix-domain X11 forwarding)

27 SSH_SMSG_X11_OPEN

32-bit int   local_channel
string       originator_string (see below)

This message can be sent by the server during the interactive session mode to indicate that a client has connected the fake X server. Local_channel is the channel number that the server has allocated for the connection. The client should try to open a
connection to the real X server, and respond with
SSH_MSG_CHANNEL_OPEN_CONFIRMATION or
SSH_MSG_CHANNEL_OPEN_FAILURE.

The field originator_string is present if both sides specified
SSH_PROTOFLAG_HOST_IN_FWD_OPEN in the protocol flags. It con-
tains a description of the host originating the connection.

28 SSH_CMSG_PORT_FORWARD_REQUEST

32-bit int  server_port
string       host_to_connect
32-bit int   port_to_connect

Sent by the client in the preparatory phase, this message
requests that server_port on the server machine be forwarded
over the secure channel to the client machine, and from there to
the specified host and port. The server should start listening
on the port, and send SSH_MSG_PORT_OPEN whenever a connection is
made to it. Supporting this message is optional, and the server
is free to reject any forward request. For example, it is
highly recommended that unless the user has been authenticated
as root, forwarding any privileged port numbers (below 1024) is
denied.

29 SSH_MSG_PORT_OPEN

32-bit int  local_channel
string       host_name
32-bit int   port
string       originator_string (see below)

Sent by either party in interactive session mode, this message
indicates that a connection has been opened to a forwarded
TCP/IP port. Local_channel is the channel number that the send-
ing party has allocated for the connection. Host_name is the
host the connection should be forwarded to, and the port is
the port on that host to connect. The receiving party should
open the connection, and respond with
SSH_MSG_CHANNEL_OPEN_CONFIRMATION or
SSH_MSG_CHANNEL_OPEN_FAILURE. It is recommended that the
receiving side check the host_name and port for validity to
avoid compromising local security by compromised remote side
software. Particularly, it is recommended that the client per-
mit connections only to those ports for which it has requested
forwarding with SSH_CMSG_PORT_FORWARD_REQUEST.

The field originator_string is present if both sides specified
SSH_PROTOFLAG_HOST_IN_FWD_OPEN in the protocol flags. It contains a description of the host originating the connection.

30 SSH_CMSG_AGENT_REQUEST_FORWARDING

(no arguments)

Requests that the connection to the authentication agent be forwarded over the secure channel. The method used by clients to contact the authentication agent within each machine is implementation and machine dependent. If the server accepts this request, it should arrange that any clients run from this session will actually contact the server program when they try to contact the authentication agent. The server should then send a SSH_SMSG_AGENT_OPEN to open a channel to the agent, and the client should forward the connection to the real authentication agent. Supporting this message is optional.

31 SSH_SMSG_AGENT_OPEN

32-bit int local_channel

Sent by the server in interactive session mode, this message requests opening a channel to the authentication agent. The client should open a channel, and respond with either SSH_MSG_CHANNEL_OPEN_CONFIRMATION or SSH_MSG_CHANNEL_OPEN_FAILURE.

32 SSH_MSG_IGNORE

string data

Either party may send this message at any time. This message, and the argument string, is silently ignored. This message might be used in some implementations to make traffic analysis more difficult. This message is not currently sent by the implementation, but all implementations are required to recognize and ignore it.

33 SSH_CMSG_EXIT_CONFIRMATION

(no arguments)

Sent by the client in response to SSH_SMSG_EXITSTATUS. This is the last message sent by the client.

34 SSH_CMSG_X11_REQUEST_FORWARDING
Sent by the client during the preparatory phase, this message requests that the server create a fake X11 display and set the DISPLAY environment variable accordingly. An internet-domain display is preferable. The given authentication protocol and the associated data should be recorded by the server so that it is used as authentication on connections (e.g., in .Xauthority). The authentication protocol must be one of the supported X11 authentication protocols, e.g., “MIT-MAGIC-COOKIE-1”. Authentication data must be a lowercase hex string of even length. Its interpretation is protocol dependent. The data is in a format that can be used with e.g. the xauth program. Supporting this message is optional.

The client is permitted (and recommended) to generate fake authentication information and send fake information to the server. This way, a corrupt server will not have access to the user’s terminal after the connection has terminated. The correct authorization codes will also not be left hanging around in files on the server (many users keep the same X session for months, thus protecting the authorization data becomes important).

X11 authentication spoofing works by initially sending fake (random) authentication data to the server, and interpreting the first packet sent by the X11 client after the connection has been opened. The first packet contains the client’s authentication. If the packet contains the correct fake data, it is replaced by the client by the correct authentication data, and then sent to the X server.

Requests authentication using /etc/hosts.equiv and .rhosts (or equivalent) together with RSA host authentication. The server should check that the client side port number is less than 1024 (a privileged port), and immediately reject authentication if it is not. The server responds with SSH_SMSG_FAILURE or SSH_SMSG_AUTH_RSA_CHALLENGE. The client must respond to the challenge with the proper SSH_CMSG_AUTH_RSA_RESPONSE. The
server then responds with success if access was granted, or failure if the client gave a wrong response. Supporting this authentication method is optional but recommended in most environments.

36 SSH_MSG_DEBUG

string debugging message sent to the other side

This message may be sent by either party at any time. It is used to send debugging messages that may be informative to the user in solving various problems. For example, if authentication fails because of some configuration error (e.g., incorrect permissions for some file), it can be very helpful for the user to make the cause of failure available. On the other hand, one should not make too much information available for security reasons. It is recommended that the client provides an option to display the debugging information sent by the sender (the user probably does not want to see it by default). The server can log debugging data sent by the client (if any). Either party is free to ignore any received debugging data. Every implementation must be able to receive this message, but no implementation is required to send these.

37 SSH_CMSG_REQUEST_COMPRESSION

32-bit int gzip compression level (1-9)

This message can be sent by the client in the preparatory operations phase. The server responds with SSH_SMSG_FAILURE if it does not support compression or does not want to compress; it responds with SSH_SMSG_SUCCESS if it accepted the compression request. In the latter case the response to this packet will still be uncompressed, but all further packets in either direction will be compressed by gzip.

Encoding of Terminal Modes

Terminal modes (as passed in SSH_CMSG_REQUEST_PTY) are encoded into a byte stream. It is intended that the coding be portable across different environments.

The tty mode description is a stream of bytes. The stream consists of opcode-argument pairs. It is terminated by opcode TTY_OP_END (0). Opcodes 1-127 have one-byte arguments. Opcodes 128-159 have 32-bit integer arguments (stored msb first). Opcodes 160-255 are not yet defined, and cause parsing to stop (they should only be used after
any other data).

The client puts in the stream any modes it knows about, and the server ignores any modes it does not know about. This allows some degree of machine-independence, at least between systems that use a POSIX-like tty interface. The protocol can support other systems as well, but the client may need to fill reasonable values for a number of parameters so the server pty gets set to a reasonable mode (the server leaves all unspecified mode bits in their default values, and only some combinations make sense).

The following opcodes have been defined. The naming of opcodes mostly follows the POSIX terminal mode flags.

0 TTY_OP_END
   Indicates end of options.

1 VINTR
   Interrupt character; 255 if none. Similarly for the other characters. Not all of these characters are supported on all systems.

2 VQUIT
   The quit character (sends SIGQUIT signal on UNIX systems).

3 VERASE
   Erase the character to left of the cursor.

4 VKILL
   Kill the current input line.

5 VEOF
   End-of-file character (sends EOF from the terminal).

6 VEOL
   End-of-line character in addition to carriage return and/or linefeed.

7 VEOL2
   Additional end-of-line character.

8 VSTART
   Continues paused output (normally ^Q).

9 VSTOP
   Pauses output (^S).

10 VSUSP
Suspends the current program.

11 VDSUSP
   Another suspend character.

12 VREPRINT
   Reprints the current input line.

13 VWERASE
   Erases a word left of cursor.

14 VLNEXT
   More special input characters; these are probably not supported on most systems.

15 VFLUSH

16 VSWTCH

17 VSTATUS

18 VDISCARD

30 IGNPAR
   The ignore parity flag. The next byte should be 0 if this flag is not set, and 1 if it is set.

31 PARMRK
   More flags. The exact definitions can be found in the POSIX standard.

32 INPCK

33 ISTRIP

34 INLCR

35 IGNCR

36 ICRNL

37 IUCLC

38 IXON

39 IXANY
40 IXOFF
41 IMAXBEL

50 ISIG
51 ICANON
52 XCASE
53 ECHO
54 ECHOE
55 ECHOK
56 ECHONL
57 NOFLSH
58 TOSTOP
59 IEXTEN
60 ECHOCTL
61 ECHOKE
62 PENDIN

70 OPOST
71 OLCUC
72 ONLCR
73 OCRNL
74 ONOCR
75 ONLRET

90 CS7
91 CS8
The Authentication Agent Protocol

The authentication agent is a program that can be used to hold RSA authentication keys for the user (in future, it might hold data for other authentication types as well). An authorized program can send requests to the agent to generate a proper response to an RSA challenge. How the connection is made to the agent (or its representative) inside a host and how access control is done inside a host is implementation-dependent; however, how it is forwarded and how one interacts with it is specified in this protocol. The connection to the agent is normally automatically forwarded over the secure channel.

A program that wishes to use the agent first opens a connection to its local representative (typically, the agent itself or an SSH server). It then writes a request to the connection, and waits for response. It is recommended that at least five minutes of timeout are provided waiting for the agent to respond to an authentication challenge (this gives sufficient time for the user to cut-and-paste the challenge to a separate machine, perform the computation there, and cut-and-paste the result back if so desired).

Messages sent to and by the agent are in the following format:

4 bytes   Length, msb first. Does not include length itself.
1 byte    Packet type. The value 255 is reserved for future extensions.
data      Any data, depending on packet type. Encoding as in the ssh packet protocol.

The following message types are currently defined:

1 SSH_AGENTC_REQUEST_RSA_IDENTITIES

(no arguments)
Requests the agent to send a list of all RSA keys for which it can answer a challenge.

2 SSH_AGENT_RSA_IDENTITIES_ANSWER

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>howmany</td>
<td>32-bit int</td>
</tr>
<tr>
<td>bits</td>
<td>32-bit int</td>
</tr>
<tr>
<td>public exponent</td>
<td>mp-int</td>
</tr>
<tr>
<td>public modulus</td>
<td>mp-int</td>
</tr>
<tr>
<td>comment</td>
<td>string</td>
</tr>
</tbody>
</table>

The agent sends this message in response to the SSH_AGENTC_REQUEST_RSA_IDENTITIES. The answer lists all RSA keys for which the agent can answer a challenge. The comment field is intended to help identify each key; it may be printed by an application to indicate which key is being used. If the agent is not holding any keys, howmany will be zero.

3 SSH_AGENTC_RSA_CHALLENGE

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>bits</td>
<td>32-bit int</td>
</tr>
<tr>
<td>public exponent</td>
<td>mp-int</td>
</tr>
<tr>
<td>public modulus</td>
<td>mp-int</td>
</tr>
<tr>
<td>challenge</td>
<td>mp-int</td>
</tr>
<tr>
<td>session_id</td>
<td>16 bytes</td>
</tr>
<tr>
<td>response_type</td>
<td>32-bit int</td>
</tr>
</tbody>
</table>

Requests RSA decryption of random challenge to authenticate the other side. The challenge will be decrypted with the RSA private key corresponding to the given public key.

The decrypted challenge must contain a zero in the highest (partial) byte, 2 in the next byte, followed by non-zero random bytes, a zero byte, and then the real challenge value in the lowermost bytes. The real challenge must be 32 8-bit bytes (256 bits).

Response_type indicates the format of the response to be returned. Currently the only supported value is 1, which means to compute MD5 of the real challenge plus session id, and return the resulting 16 bytes in a SSH_AGENT_RSA_RESPONSE message.

4 SSH_AGENT_RSA_RESPONSE

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD5 of decrypted challenge</td>
<td>16 bytes</td>
</tr>
</tbody>
</table>

Answers an RSA authentication challenge. The response is 16
bytes: the MD5 checksum of the 32-byte challenge.

5 SSH_AGENT_FAILURE
(no arguments)

This message is sent whenever the agent fails to answer a request properly. For example, if the agent cannot answer a challenge (e.g., no longer has the proper key), it can respond with this. The agent also responds with this message if it receives a message it does not recognize.

6 SSH_AGENT_SUCCESS
(no arguments)

This message is sent by the agent as a response to certain requests that do not otherwise cause a message be sent. Currently, this is only sent in response to SSH_AGENTC_ADD_RSA_IDENTITY and SSH_AGENTC_REMOVE_RSA_IDENTITY.

7 SSH_AGENTC_ADD_RSA_IDENTITY

32-bit int bits
mp-int public modulus
mp-int public exponent
mp-int private exponent
mp-int multiplicative inverse of p mod q
mp-int p
mp-int q
string comment

Registers an RSA key with the agent. After this request, the agent can use this RSA key to answer requests. The agent responds with SSH_AGENT_SUCCESS or SSH_AGENT_FAILURE.

8 SSH_AGENTC_REMOVE_RSA_IDENTITY

32-bit int bits
mp-int public exponent
mp-int public modulus

Removes an RSA key from the agent. The agent will no longer accept challenges for this key and will not list it as a supported identity. The agent responds with SSH_AGENT_SUCCESS or SSH_AGENT_FAILURE.

If the agent receives a message that it does not understand, it
responds with SSH_AGENT_FAILURE. This permits compatible future extensions.

It is possible that several clients have a connection open to the authentication agent simultaneously. Each client will use a separate connection (thus, any SSH connection can have multiple agent connections active simultaneously).

References


[GZIP]
The GNU GZIP program; available for anonymous ftp at prep.ai.mit.edu. Please let me know if you know a paper describing the algorithm.

[IDEA]

[PKCS#1]

[POSIX]

[RFC0791]

[RFC0793]
Security Considerations

This protocol deals with the very issue of user authentication and security.

First of all, as an implementation issue, the server program will have to run as root (or equivalent) on the server machine. This is because the server program will need be able to change to an arbitrary user id. The server must also be able to create a privileged TCP/IP port.

The client program will need to run as root if any variant of .rhosts authentication is to be used. This is because the client program will need to create a privileged port. The client host key is also usually stored in a file which is readable by root only. The client needs the host key in .rhosts authentication only. Root privileges can be dropped as soon as the privileged port has been created and the host key has been read.

The SSH protocol offers major security advantages over existing telnet and rlogin protocols.

- IP spoofing is restricted to closing a connection (by encryption, host keys, and the special random cookie). If encryption is not used, IP spoofing is possible for those who can hear packets going out from the server.

- DNS spoofing is made ineffective (by host keys).

- Routing spoofing is made ineffective (by host keys).

- All data is encrypted with strong algorithms to make...
eavesdropping as difficult as possible. This includes encrypting any authentication information such as passwords. The information for decrypting session keys is destroyed every hour.

- Strong authentication methods: .rhosts combined with RSA host authentication, and pure RSA authentication.
- X11 connections and arbitrary TCP/IP ports can be forwarded securely.
- Man-in-the-middle attacks are deterred by using the server host key to encrypt the session key.
- Trojan horses to catch a password by routing manipulation are deterred by checking that the host key of the server machine matches that stored on the client host.

The security of SSH against man-in-the-middle attacks and the security of the new form of .rhosts authentication, as well as server host validation, depends on the integrity of the host key and the files containing known host keys.

The host key is normally stored in a root-readable file. If the host key is compromised, it permits attackers to use IP, DNS and routing spoofing as with current rlogin and rsh. It should never be any worse than the current situation.

The files containing known host keys are not sensitive. However, if an attacker gets to modify the known host key files, it has the same consequences as a compromised host key, because the attacker can then change the recorded host key.

The security improvements obtained by this protocol for X11 are of particular significance. Previously, there has been no way to protect data communicated between an X server and a client running on a remote machine. By creating a fake display on the server, and forwarding all X11 requests over the secure channel, SSH can be used to run any X11 applications securely without any cooperation with the vendors of the X server or the application.

Finally, the security of this program relies on the strength of the underlying cryptographic algorithms. The RSA algorithm is used for authentication key exchange. It is widely believed to be secure. Of the algorithms used to encrypt the session, DES has a rather small key these days, probably permitting governments and organized criminals to break it in very short time with specialized hardware. 3DES is probably safe (but slower). IDEA is widely believed to be secure. People have varying degrees of confidence in the other algorithms.
This program is not secure if used with no encryption at all.

Additional Information

Additional information (especially on the implementation and mailing lists) is available via WWW at http://www.cs.hut.fi/ssh.

Comments should be sent to Tatu Ylonen <ylo@cs.hut.fi> or the SSH Mailing List <ssh@cclinet.fi>.

Author’s Address

Tatu Ylonen
Helsinki University of Technology
Otakaari 1
FIN-02150 Espoo, Finland

Phone: +358-0-451-3374
Fax: +358-0-451-3293
EMail: ylo@cs.hut.fi