SSH Connection Protocol

T. Ylonen, T. Kivinen, M. Saarinen, T. Rinne and S. Lehtinen 6 August 1998

SSH is a protocol for secure remote login and other secure network services over an insecure network. This document describes the SSH connection protocol. It provides interactive login sessions, remote execution of commands, forwarded TCP/IP connections, and forwarded X11 connections. All of these channels are multiplexed into a single encrypted tunnel. The SSH Connection Protocol has been designed to run on top of the SSH transport layer and user authentication protocols.

Contents

1 Introduction

The SSH Connection Protocol has been designed to run on top of the SSH transport layer and user authentication protocols. It provides interactive login sessions, remote execution of commands, forwarded TCP/IP connections, and forwarded X11 connections. The service name for this protocol (after user authentication) is "ssh-connection".

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

2 Global Requests

There are several kinds of requests that affect the state of the remote end "globally", independent of any channels. An example is a request to start TCP/IP forwarding for a specific port. All such requests use the following format.

```
byte      SSH_MSG_GLOBAL_REQUEST
string    request name (restricted to US-ASCII)
boolean   want reply
... request-specific data follows
```

The recipient will respond to this message with SSH_MSG_REQUEST_SUCCESS, SSH_MSG_REQUEST_FAILURE, or some request-specific continuation messages if '`want reply`' is TRUE.

```
byte      SSH_MSG_REQUEST_SUCCESS
```

If the recipient does not recognize or support the request, it simply responds with SSH_MSG_REQUEST_FAILURE.

```
byte      SSH_MSG_REQUEST_FAILURE
```

3 Channel Mechanism

All terminal sessions, forwarded connections, etc. are channels. Either side may open a channel. Multiple channels are multiplexed into a single connection.

Channels are identified by numbers at each end. The number referring to a channel may be different on each side. Requests to open a channel contain the sender's channel number. Any other channel-related messages contain the recipient's channel number for the channel.

Channels are flow-controlled. No data may be sent to a channel until a message is received to indicate that window space is available.

3.1 Opening a Channel

When either side wishes to open a new channel, it allocates a local number for the channel. It then sends the following message to the other side, and includes the local channel number and initial window size in the message.

```
byte      SSH_MSG_CHANNEL_OPEN
string    channel type (restricted to US-ASCII)
uint32    sender channel
uint32    initial window size
uint32    maximum packet size
... channel type specific data follows
```

The channel type is a name as described in the SSH architecture document, with similar extension mechanisms. '`sender channel`' is a local identifier for the channel used by the sender of this message. '`initial window size`' specifies how many bytes of channel data can be sent to the sender of this message without adjusting the window. '`Maximum packet size`' specifies the maximum size of an individual data packet that can be sent to the sender (for example, one might want to use smaller packets for interactive connections to get better interactive response on slow links).

The remote side then decides whether it can open the channel, and responds with either

```
byte      SSH_MSG_CHANNEL_OPEN_CONFIRMATION
uint32    recipient channel
uint32    sender channel
uint32    initial window size
uint32    maximum packet size
... channel type specific data follows
```

where '`recipient channel`' is the channel number given in the original open request, and '`sender channel`' is the channel number allocated by the other side, or

```
byte      SSH_MSG_CHANNEL_OPEN_FAILURE
uint32    recipient channel
uint32    reason code
string    additional textual information (ISO-10646 UTF-8
          [RFC-2044])
string    language tag (as defined in [RFC-1766])
```

If the recipient of the SSH_MSG_CHANNEL_OPEN message does not support the specified channel type, it simply responds with SSH_MSG_CHANNEL_OPEN_FAILURE. The client MAY show the additional information to the user. If this is done, the client software should take the precautions discussed in [SSH-ARCH].

The following reason codes are defined:

```
#define SSH_OPEN_ADMINISTRATIVELY_PROHIBITED    1
#define SSH_OPEN_CONNECT_FAILED                 2
#define SSH_OPEN_UNKNOWN_CHANNEL_TYPE           3
#define SSH_OPEN_RESOURCE_SHORTAGE              4
```

3.2 Data Transfer

The window size specifies how many bytes the other party can send before it must wait for the window to be adjusted. Both parties use the following message to adjust the window.

```
  byte      SSH_MSG_CHANNEL_WINDOW_ADJUST
  uint32    recipient channel
  uint32    bytes to add
```

After receiving this message, the recipient MAY send the given number of bytes more that it was previously allowed to send; the window size is incremented.

Data transfer is done with messages of the following type.

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

The maximum amount of data allowed is the current window size. The window size is decremented by the amount of data sent. Both parties MAY ignore all extra data sent after the allowed window is empty.

Additionally, some channels can transfer several types of data. An example of this is `stderr` data from interactive sessions. Such data can be passed with SSH_MSG_CHANNEL_EXTENDED_DATA messages, where a separate integer specifies the type of the data. The available types and their interpretation depend on the type of the channel.

```
  byte      SSH_MSG_CHANNEL_EXTENDED_DATA
  uint32    recipient_channel
  uint32    data_type_code
  string    data
```

Data sent with these messages consumes the same window as ordinary data.

Currently, only the following type is defined.

```
#define SSH_EXTENDED_DATA_STDERR                1
```

3.3 Closing a Channel

When a party will no longer send more data to a channel, it SHOULD send SSH_MSG_CHANNEL_EOF.

```
byte      SSH_MSG_CHANNEL_EOF
uint32    recipient_channel
```

No explicit response is sent to this message; however, the application may send EOF to whatever is at the other end of the channel. Note that the channel remains open after this message, and more data may still be sent in the other direction. This message does not consume window space and can be sent even if no window space is available.

When either party wishes to terminate the channel, it sends SSH_MSG_CHANNEL_CLOSE. Upon receiving this message, a party MUST send back a SSH_MSG_CHANNEL_CLOSE unless it has already sent this message for the channel. The channel is considered closed for a party when it has both sent and received SSH_MSG_CHANNEL_CLOSE, and the party may then reuse the channel number. A party MAY send SSH_MSG_CHANNEL_CLOSE without having sent or received SSH_MSG_CHANNEL_EOF.

```
byte      SSH_MSG_CHANNEL_CLOSE
uint32    recipient_channel
```

This message does not consume window space and can be sent even if no window space is available.

It is recommended that any data sent before this message is delivered to the actual destination, if possible.

3.4 Channel-Specific Requests

Many channel types have extensions that are specific to that particular channel type. An example is requesting a pty (pseudo terminal) for an interactive session.

All channel-specific requests use the following format.

```
byte      SSH_MSG_CHANNEL_REQUEST
uint32    recipient channel
string    request type (restricted to US-ASCII)
boolean   want reply
... type-specific data
```

If want reply is FALSE, no response will be sent to the request. Otherwise, the recipient responds with either SSH_MSG_CHANNEL_SUCCESS or SSH_MSG_CHANNEL_FAILURE, or request-specific continuation messages. If the request is not recognized or is not supported for the channel, SSH_MSG_CHANNEL_FAILURE is returned.

This message does not consume window space and can be sent even if no window space is available. Request types are local to each channel type.

The client is allowed to send further messages without waiting for the response to the request.

```
byte      SSH_MSG_CHANNEL_SUCCESS
uint32    recipient_channel

byte      SSH_MSG_CHANNEL_FAILURE
uint32    recipient_channel
```

These messages do not consume window space and can be sent even if no window space is available.

4 Interactive Sessions

A session is a remote execution of a program. The program may be a shell, an application, a system command, or some built-in subsystem. It may or may not have a tty, and may or may not involve X11 forwarding. Multiple sessions can be active simultaneously.

4.1 Opening a Session

A session is started by sending the following message.

```
byte      SSH_MSG_CHANNEL_OPEN
string    "session"
uint32    sender channel
uint32    initial window size
uint32    maximum packet size
```

Client implementations SHOULD reject any session channel open requests to make it more difficult for a corrupt server to attack the client.

4.2 Requesting a Pseudo-Terminal

A pseudo-terminal can be allocated for the session by sending the following message.

```
byte      SSH_MSG_CHANNEL_REQUEST
uint32    recipient_channel
string    "pty-req"
boolean   want_reply
string    TERM environment variable value (e.g., vt100)
uint32    terminal width, characters (e.g., 80)
uint32    terminal height, rows (e.g., 24)
uint32    terminal width, pixels (e.g., 480)
uint32    terminal height, pixels (e.g., 640)
string    encoded terminal modes
```

The encoding of terminal modes is described in Section 6 (Encoding of Terminal Modes). Zero dimension parameters MUST be ignored. The character/row dimensions override the pixel dimensions (when nonzero). Pixel dimensions refer to the drawable area of the window.

The dimension parameters are only informational.

The client SHOULD ignore pty requests.

4.3 X11 Forwarding

4.3.1 Requesting X11 Forwarding

X11 forwarding may be requested for a session by sending

```
byte      SSH_MSG_CHANNEL_REQUEST
uint32    recipient channel
```

```
string    "x11-req"
boolean   want reply
boolean   single connection
string    x11 authentication protocol
string    x11 authentication cookie
uint32    x11 screen number
```

It is recommended that the authentication cookie that is sent be a fake, random cookie, and that the cookie is checked and replaced by the real cookie when a connection request is received.

X11 connection forwarding should stop when the session channel is closed; however, already opened forwardings should not be automatically closed when the session channel is closed.

If '`single connection`' is TRUE, only a single connection should be forwarded. No more connections will be forwarded after the first, or after the session channel has been closed.

'`X11 authentication protocol` is the name of the X11 authentication method used, i.e. "MIT-MAGIC-COOKIE-1".

4.3.2 X11 Channels

X11 channels are opened with a channel open request. The resulting channels are independent of the session, and closing the session channel does not close the forwarded X11 channels.

```
byte      SSH_MSG_CHANNEL_OPEN
string    "x11"
uint32    sender channel
uint32    initial window size
uint32    maximum packet size
string    originator address (e.g. "192.168.7.38")
uint32    originator port
```

The recipient should respond with SSH_MSG_CHANNEL_OPEN_CONFIRMATION or SSH_MSG_CHANNEL_OPEN_FAILURE.

Implementations MUST reject any X11 channel open requests if they have not requested X11 forwarding.

4.4 Authentication Agent Forwarding

It is RECOMMENDED that authentication agent forwarding is allowed even when either or both parties do not support the SSH authentication agent protocol [SSH-AGENT].

4.4.1 Requesting Authentication Agent Forwarding

Authentication agent forwarding may be requested for a session by sending

```
byte      SSH_MSG_CHANNEL_REQUEST
uint32    recipient channel
string    "auth-agent-req"
boolean   want reply
```

The server responds with either SSH_MSG_CHANNEL_SUCCESS or SSH_MSG_CHANNEL_FAILURE (if '`want reply`' is TRUE). The client MAY send further messages without waiting for the response to this message.

4.4.2 Authentication Agent Channels

When an application requests a connection to the authentication agent, the following message is sent to the originator of the session.

```
byte      SSH_MSG_CHANNEL_OPEN
string    "auth-agent"
uint32    sender channel
uint32    initial window size
uint32    maximum packet size
```

The recipient SHOULD respond with open confirmation or open failure.

Implementations MUST reject any agent channel open requests if they have not requested agent forwarding.

4.5 SSH1 Authentication Agent Forwarding

Implementations MAY support ssh1 authentication agent forwarding in order to provide compatibility with old ssh versions.

4.5.1 Requesting SSH1 Authentication Agent Forwarding

Authentication agent forwarding may be requested for a session by sending

```
byte      SSH_MSG_CHANNEL_REQUEST
uint32    recipient channel
string    "auth-ssh1-agent-req"
boolean   want reply
```

The server responds with either SSH_MSG_CHANNEL_SUCCESS or SSH_MSG_CHANNEL_FAILURE (if '`want reply`' is TRUE). The client MAY send further messages without waiting for the response to this message.

4.5.2 SSH1 Authentication Agent Channels

When an application requests a connection to the authentication agent, the following message is sent to the originator of the session.

```
byte      SSH_MSG_CHANNEL_OPEN
string    "auth-ssh1-agent"
uint32    sender channel
uint32    initial window size
uint32    maximum packet size
```

The recipient SHOULD respond with open confirmation or open failure.

Implementations MUST reject any agent channel open requests if they have not requested ssh1 agent forwarding.

4.6 Environment Variable Passing

Environment variables may be passed to the shell/command to be started later. Typically, each machine will have a preconfigured set of variables that it will allow. Since uncontrolled setting of environment variables can be very dangerous, it is recommended that implementations allow setting only variables whose names have been explicitly configured to be allowed.

```
byte      SSH_MSG_CHANNEL_REQUEST
uint32    recipient channel
string    "env"
boolean   want reply
string    variable name
string    variable value
```

4.7 Starting a Shell or a Command

Once the session has been set up, a program is started at the remote end. Program can be a shell, an application program or a subsystem with a host-independent name. Only one of these requests can succeed per channel.

```
byte      SSH_MSG_CHANNEL_REQUEST
uint32    recipient channel
string    "shell"
boolean   want reply
```

This message will request the user's default shell (typically defined in /etc/passwd in UNIX systems) to be started at the other end.

```
byte      SSH_MSG_CHANNEL_REQUEST
uint32    recipient channel
string    "exec"
boolean   want reply
string    command
```

This message will request the server to start the execution of the given command. The command string may contain a path. Normal precautions MUST be taken to prevent the execution of unauthorized commands.

```
byte      SSH_MSG_CHANNEL_REQUEST
uint32    recipient channel
string    "subsystem"
boolean   want reply
string    subsystem name
```

This last form executes a predefined subsystem. It expected that these will include a general file transfer mechanism, and possibly other features. Implementations may also allow configuring more such mechanisms.

The server SHOULD not halt the execution of the protocol stack when starting a shell or a program. All input and output from these SHOULD be redirected the the channel or to the encrypted tunnel.

It is RECOMMENDED to request and check the reply for these messages. The client SHOULD ignore these messages.

4.8 Session Data Transfer

Data transfer for a session is done using SSH_MSG_CHANNEL_DATA and SSH_MSG_CHANNEL_EXTENDED_DATA packets and the window mechanism. The extended data type SSH_EXTENDED_DATA_STDERR has been defined for `stderr` data.

4.9 Window Dimension Change Message

When the window (terminal) size changes on the client side, it MAY send a message to the other side to inform it of the new dimensions.

```
byte      SSH_MSG_CHANNEL_REQUEST
uint32    recipient_channel
string    "window-change"
boolean   FALSE
uint32    terminal width, columns
uint32    terminal height, rows
uint32    terminal width, pixels
uint32    terminal height, pixels
```

No response SHOULD be sent to this message.

4.10 Local Flow Control

On many systems it is possible to determine if a pseudo-terminal is using control-S control-Q flow control. When flow control is allowed, it is often desirable to do the flow control at the client end to speed up responses to user requests. This is facilitated by the following notification. Initially, the server is responsible for flow control. (Here, again, client means the side originating the session, and server the other side.)

The message below is used by the server to inform the client when it can or cannot perform flow control (control-S/control-Q processing). If '`client can do`' is TRUE, the client is allowed to do flow control using control-S and control-Q. The client MAY ignore this message.

```
byte      SSH_MSG_CHANNEL_REQUEST
uint32    recipient channel
string    "xon-xoff"
boolean   FALSE
boolean   client can do
```

No response is sent to this message.

4.11 Signals

A signal can be delivered to the remote process/service using the following message. Some systems may not implement signals, in which case they SHOULD ignore this message.

```
byte      SSH_MSG_CHANNEL_REQUEST
uint32    recipient channel
string    "signal"
boolean   FALSE
uint32    signal number
```

4.12 Returning Exit Status

When the command running at the other end terminates, The following message can be sent to return the exit status of the command. Returning the status is RECOMMENDED. No acknowledgment is sent for this message. The channel needs to be closed with SSH_MSG_CHANNEL_CLOSE after this message.

The client MAY ignore these messages.

```
byte      SSH_MSG_CHANNEL_REQUEST
uint32    recipient_channel
string    "exit-status"
boolean   FALSE
uint32    exit_status
```

The remote command may also terminate violently due to a signal. Such a condition can be indicated by the following message. A zero `exit_status` usually means that the command terminated successfully.

```
byte      SSH_MSG_CHANNEL_REQUEST
uint32    recipient channel
string    "exit-signal"
boolean   FALSE
uint32    signal number
boolean   core dumped
string    error message (ISO-10646 UTF-8 [RFC-2044])
string    language tag (as defined in [RFC-1766])
```

The '`error message`' contains an additional explanation of the error message. The message may consist of multiple lines. The client software MAY display this message to the user.

5 TCP/IP Port Forwarding

5.1 Requesting Port Forwarding

A party need not explicitly request forwardings from its own end to the other direction. However, it if wishes to have connections to a port on the other side be forwarded to the local side, it must explicitly request this.

```
byte      SSH_MSG_GLOBAL_REQUEST
string    "tcpip-forward"
boolean   want reply
string    address to bind (e.g. "0.0.0.0")
uint32    port number to bind
```

'`Address to bind`' and '`port number to bind`' specify the IP address and port to which the socket to be listened is bound. The address should be "0.0.0.0" if connections are allowed from anywhere. (Note that the client can still filter connections based on information passed in the open request.)

Implementations should only allow forwarding privileged ports if the user has been authenticated as a privileged user.

Client implementations SHOULD reject these messages; they are normally only sent by the client.

A port forwarding can be cancelled with the following message. Note that channel open requests may be received until a reply to this message is received.

```
byte      SSH_MSG_GLOBAL_REQUEST
string    "cancel-tcpip-forward"
boolean   want reply
string    address_to_bind (e.g. "127.0.0.1")
uint32    port number to bind
```

Client implementations SHOULD reject these messages; they are normally only sent by the client.

5.2 TCP/IP Forwarding Channels

When a connection comes to a port for which remote forwarding has been requested, a channel is opened to forward the port to the other side.

```
byte      SSH_MSG_CHANNEL_OPEN
string    "forwarded-tcpip"
uint32    sender channel
uint32    initial window size
uint32    maximum packet size
string    address that was connected
uint32    port that was connected
string    originator IP address
uint32    originator port
```

Implementations MUST reject these messages unless they have previously requested a remote TCP/IP port forwarding with the given port number.

When a connection comes to a locally forwarded TCP/IP port, the following packet is sent to the other side. Note that these messages MAY be sent also for ports for which no forwarding has been explicitly requested. The receiving side must decide whether to allow the forwarding.

```
byte      SSH_MSG_CHANNEL_OPEN
string    "direct-tcpip"
uint32    sender channel
uint32    initial window size
uint32    maximum packet size
string    host to connect
uint32    port to connect
string    originator IP address
uint32    originator port
```

'`Host to connect`' and '`port to connect`' specify the TCP/IP host and port where the recipient should connect the channel. '`Host to connect`' may be either a domain name or a numeric IP address.

'`Originator IP address`' is the numeric IP address of the machine where the connection request comes from, and '`originator port`' is the port on the originator host from where the connection came from.

Forwarded TCP/IP channels are independent of any sessions, and closing a session channel does not in any way imply that forwarded connections should be closed.

Client implementations SHOULD reject direct TCP/IP open requests for security reasons.

6 Encoding of Terminal Modes

Terminal modes (as passed in a pty request) 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 1-159 have a single uint32 argument. Opcodes 160-255 are not yet defined, and cause parsing to stop (they should only be used after any other data).

The client SHOULD put in the stream any modes it knows about, and the server MAY ignore 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 POSIX
                   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 control-Q).
9   VSTOP          Pauses output (normally control-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         Enter the next character typed literally, even if it
                   is a special character
15  VFLUSH         Character to flush output.
16  VSWTCH         Switch to a different shell layer.
17  VSTATUS        Prints system status line (load, command, pid etc).
18  VDISCARD       Toggles the flushing of terminal output.
30  IGNPAR         The ignore parity flag.  The parameter SHOULD be 0 if
                   this flag is FALSE set, and 1 if it is TRUE.
31  PARMRK         Mark parity and framing errors.
32  INPCK          Enable checking of parity errors.
33  ISTRIP         Strip 8th bit off characters.
34  INLCR          Map NL into CR on input.
35  IGNCR          Ignore CR on input.
36  ICRNL          Map CR to NL on input.
37  IUCLC          Translate uppercase characters to lowercase.
38  IXON           Enable output flow control.
```

```
39  IXANY         Any char will restart after stop.
40  IXOFF         Enable input flow control.
41  IMAXBEL       Ring bell on input queue full.
50  ISIG          Enable signals INTR, QUIT, [D]SUSP.
51  ICANON        Canonicalize input lines.
52  XCASE         Enable input and output of uppercase characters by
                  preceding their lowercase equivalents with ‘\\’.
53  ECHO          Enable echoing.
54  ECHOE         Visually erase chars.
55  ECHOK         Kill character discards current line.
56  ECHONL        Echo NL even if ECHO is off.
57  NOFLSH        Don’t flush after interrupt.
58  TOSTOP        Stop background jobs from output.
59  IEXTEN        Enable extensions.
60  ECHOCTL       Echo control characters as ^(Char).
61  ECHOKE        Visual erase for line kill.
62  PENDIN        Retype pending input.
70  OPOST         Enable output processing.
71  OLCUC         Convert lowercase to uppercase.
72  ONLCR         Map NL to CR-NL.
73  OCRNL         Translate carriage return to newline (output).
74  ONOCR         Translate newline to carriage return-newline
                  (output).
75  ONLRET        Newline performs a carriage return (output).
90  CS7           7 bit mode.
91  CS8           8 bit mode.
92  PARENB        Parity enable.
93  PARODD        Odd parity, else even.

128 TTY_OP_ISPEED  Specifies the input baud rate in bits per second.
129 TTY_OP_OSPEED  Specifies the output baud rate in bits per second.
```

7 Summary of Message Numbers

```
#define SSH_MSG_GLOBAL_REQUEST                  80
#define SSH_MSG_REQUEST_SUCCESS                 81
#define SSH_MSG_REQUEST_FAILURE                 82
#define SSH_MSG_CHANNEL_OPEN                    90
#define SSH_MSG_CHANNEL_OPEN_CONFIRMATION       91
#define SSH_MSG_CHANNEL_OPEN_FAILURE            92
#define SSH_MSG_CHANNEL_WINDOW_ADJUST           93
#define SSH_MSG_CHANNEL_DATA                    94
#define SSH_MSG_CHANNEL_EXTENDED_DATA           95
#define SSH_MSG_CHANNEL_EOF                     96
#define SSH_MSG_CHANNEL_CLOSE                   97
#define SSH_MSG_CHANNEL_REQUEST                 98
#define SSH_MSG_CHANNEL_SUCCESS                 99
#define SSH_MSG_CHANNEL_FAILURE                 100
```

8 Security Considerations

This protocol is assumed to run on top of a secure, authenticated transport. User authentication and protection against network-level attacks are assumed to be provided by the underlying protocols.

This protocol can, however, be used to execute commands on remote machines. The protocol also permits the server to run commands on the client. Implementations may wish to disallow this to prevent an attacker from coming from the server machine to the client machine.

X11 forwarding provides major security improvements over normal cookie-based X11 forwarding. The cookie never needs to be transmitted in the clear, and traffic is encrypted and integrity-protected. No useful authentication data will remain on the server machine after the connection has been closed. On the other hand, in some situations a forwarded X11 connection might be used to get access to the local X server across security perimeters.

Port forwardings can potentially allow an intruder to cross security perimeters such as firewalls. They do not offer anything fundamentally new that a user couldn't do otherwise; however, they make opening tunnels very easy. Implementations should allow policy control over what can be forwarded. Administrators should be able to deny forwardings where appropriate.

Since this protocol normally runs inside an encrypted tunnel, firewalls will not be able to examine the traffic.

It is RECOMMENDED that implementations disable all of the potentially dangerous features (e.g. agent forwarding, X11 forwarding, and TCP/IP forwarding) if the host key has changed.

9 References

[RFC-1766] Alvestrand, H., "Tags for the Identification of Languages", March 1995.

[RFC-1884] Hinden, R., and Deering, S., "IP Version 6 Addressing Architecture", December 1995

[RFC-2044] Yergeau, F., "UTF-8, a Transformation Format of Unicode and ISO 10646", October 1996.

[SSH-ARCH] Ylonen, T., Kivinen, T, and Saarinen, M., "SSH Protocol Architecture", Internet Draft, draft-ietf-secsh-architecture-00.txt

[SSH-TRANS] Ylonen, T., Kivinen, T, and Saarinen, M., "SSH Transport Layer Protocol", Internet Draft, draft-ietf-secsh-transport-02.txt

[SSH-USERAUTH] Ylonen, T., Kivinen, T, and Saarinen, M., "SSH Authentication Protocol", Internet Draft, draft-ietf-secsh-userauth-02.txt

10 Authors' Addresses

Tatu Ylonen
SSH Communications Security Ltd.
Tekniikantie 12
FIN-02150 ESPOO
Finland
E-mail: ylo@ssh.fi

Tero Kivinen
SSH Communications Security Ltd.
Tekniikantie 12

FIN-02150 ESPOO
Finland
E-mail: kivinen@ssh.fi

Markku-Juhani O. Saarinen
SSH Communications Security Ltd.
Tekniikantie 12
FIN-02150 ESPOO
Finland
E-mail: mjos@ssh.fi

Timo J. Rinne
SSH Communications Security Ltd.
Tekniikantie 12
FIN-02150 ESPOO
Finland
E-mail: tri@ssh.fi

Sami Lehtinen
SSH Communications Security Ltd.
Tekniikantie 12
FIN-02150 ESPOO
Finland
E-mail: sjl@ssh.fi