7

SSH

7.1 Introduction

Historically, `telnet`, `ftp`, and the BSD r-commands (`rcp`, `rsh`, `rexec`, and `rlogin`) have been used to handle interactive sessions and file transfers between a local and a remote host. Although these utilities are still popular and in widespread use, their severe security problems make them unsuitable for use in settings where security is a concern. For example, `telnet` and `ftp` provide no encryption or authentication services, so any data transferred using them is vulnerable to eavesdroppers using simple passive attacks. More seriously, these utilities send the user's password as plaintext, allowing the attacker to recover it and subsequently log on to the remote system as the user.

The r-commands are even worse. They share the same problems as `telnet` and `ftp` and are often configured to use a convenience mode that does not require the user to present any credentials. This mode is easily subverted to allow an attacker on any machine to log on to or run commands on the target machine as any user authorized to use the r-commands.

The *Secure Shell* (SSH) suite is a set of programs that serve as drop-in replacements for `telnet`, `ftp`, and the r-commands.

> More accurately, SSH is a set of protocols. Because their most popular implementations are the UNIX programs `ssh` and `sshd`, most users think of SSH as its implementation rather than the underlying protocols.

Despite its name, SSH has nothing to do with a shell, such as `sh`, `csh`, or `bash`. Rather, SSH provides a secure connection over which a user may, among other things, run a remote shell session.

When we examine this connection, we will see that it meets our requirements for a VPN. Data sent, for example, over the public Internet is encrypted and authenticated, ensuring that it is safe from snooping and alteration. From the user's perspective, these VPN functions are transparent. The user need merely call `ssh` rather than, say, `rsh` to enjoy the benefits of VPN-like security.

Two versions of SSH are in use today. These are not *program* versions; they are *protocol* versions. That is, the SSH protocol has two independent versions. Fortunately, most implementations support both versions and will negotiate which version to use at session start-up time.

In 1995, Helsinki University of Technology researcher Tatu Ylönen developed the first version of SSH. As often happens, he designed it for his own use, in this case, as a response to a password-sniffing attack on his university's network. As also often happens, Ylönen released his code for others to benefit from, and its use exploded all over the world. To deal with the increasing support issues, Ylönen formed SSH Communications Security (SCS, <`http://www.ssh.com`>) that same year. This version of the software is now known as SSH version 1 (SSHv1).

Actually version 1 of the protocol underwent steady refinement. What is now known as SSHv1 is really version 1.5 of the protocol.

As with SSL, there were no formal design documents for the first version of SSH, but Ylönen did document the protocol after the fact as an Internet Draft (draft-ylonen-ssh-protocol-00.txt). This draft has long since expired, of course, but is still distributed with the SSH source code and is available in various repositories on the Web (see, for example, <`http://www.watersprings.org/pub/id/index-y.html`>).

Because of security problems with SSHv1, SCS released version 2 of the protocol in 1996. SSHv2 is a complete rewrite of the SSH protocol and is incompatible with SSHv1. The IETF became involved by forming the Secure Shell working group (SECSH). Their Web site is at <`http://www.ietf.org/html.charters/secsh-charter.html`>.

In late 1999, in response to increasingly restrictive licenses from SCS, the OpenSSH project (<`http://www.openssh.com`>) released an SSHv1 implementation based on SCS's 1.2.12 release. This version supported protocol versions 1.3 and 1.5. In June 2000, OpenSSH released support for SSHv2, and support for Secure FTP (SFTP) followed soon afterward in November of that year. At this time, the OpenSSH suite is the most common implementation of the SSH protocols.

7.2 The SSHv1 Protocol

Like SSL, SSH is a transport-layer protocol and uses TCP to carry its packets. This has the usual advantages of providing an underlying reliable transport, freeing SSH from having to worry about retransmissions, packet ordering, and flow control. Unlike SSL, SSH does not require that either the local or remote application be SSH-aware. The situation is more analogous to an `stunnel` environment, such as that in Figure 6.24. That is, SSH provides a secure tunnel through which local and remote applications may communicate.

The most common case is shown in Figure 7.1: A local user is communicating with a remote shell. In this case, the `ssh` client is providing the local user with a terminal interface, but this is merely a convenience. This use of `ssh` is as a secure replacement for `rsh` and is virtually identical from the user's perspective.

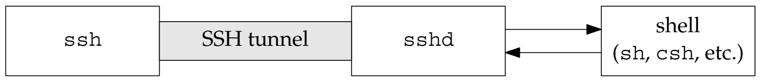

Figure 7.1 SSH as a Remote Shell

Because the use of `ssh` as a replacement for `rsh` and `telnet` is so common, we first examine the SSH protocol from the point of view of the remote shell application. Later, we consider other applications and capabilities of the SSH protocols.

Let's start with a simple interactive session and watch the protocol in action as SSH connects; authenticates the server, client, and user; transfers user data securely; and finally disconnects. In order to see the unencrypted packets, we specify null encryption (`-c none`). As we see, `ssh` warns us that there will be no encryption and that the password will be passed in the clear, just as it is for, say, `telnet`.

> Although the SSH protocol recommends that null encryption should be available for debugging purposes and although OpenSSH does provide support for it, there is no way to request it from the command line. We are using a patched version that recognizes the `-c none` option. Our patched server is listening on port 2022 instead of the normal port 22; that is why we specify `-p 2022` on the call to `ssh`.

The `-1` and `-4` specify the version 1 protocol and IPv4, respectively.

```
$ ./ssh -1 -4 -c none -p 2022 guest@localhost
WARNING: Encryption is disabled! Password will be transmitted
in clear text.
guest@localhost's password:
Last login: Sat May 15 14:55:16 2004 from localhost
Have a lot of fun...
guest@linuxlt:~> ls
Documents  public_html
guest@linuxlt:~> exit
logout
Connection to localhost closed.
```

After we supply our password, `sshd` starts a shell for us, and we list the home directory of user guest. Finally, we exit from the shell, and the connection is torn down.

Before studying the protocol messages for this session in detail, we must examine the SSHv1 binary protocol packet. Figure 7.2 shows the format of these packets.

> As with the SSL packets from Chapter 6, the SSH packet does not necessarily align its data on word boundaries, so we display them as we did for SSL.

The *length* field is the size of the packet, not including the length field itself or the variable-length *random padding* field that follows it. The padding field is intended to

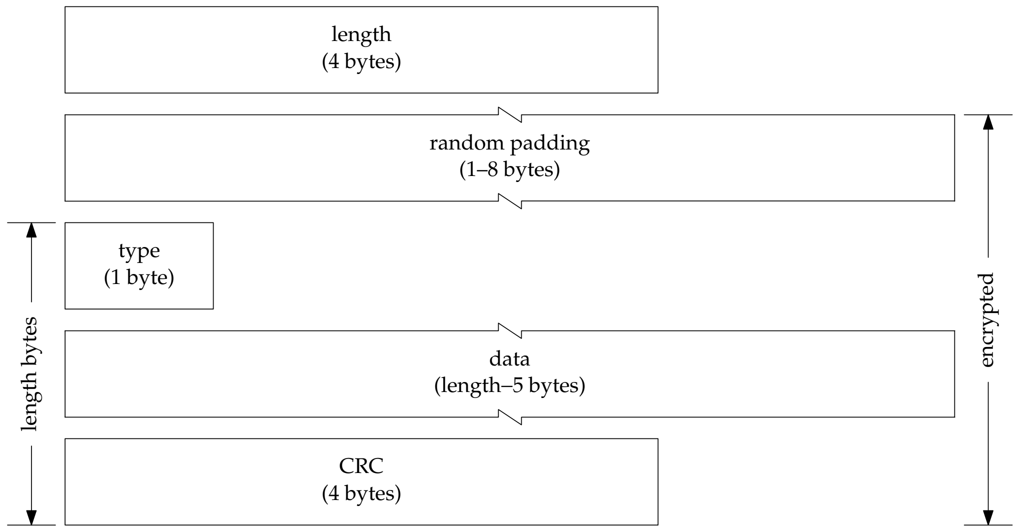

Figure 7.2 The SSHv1 Binary Packet

make known text attacks more difficult. Its size is chosen to make the size of the encrypted part of the packet a multiple of 8 bytes.

This is presumably because all the block ciphers originally supported by SSHv1 have a 64-bit block size. The OpenSSH version 1 protocol supports only DES, 3DES, and Blowfish, all of which use a 64-bit block size.

Following the padding is a 1-byte *type* field that identifies the type of message that the packet contains. The type values are shown in Figure 7.3.

The type field is followed by the message *data*. The *CRC* field, which serves as a MAC, ends the packet. When encryption is enabled, everything except the length field is encrypted.

The protocol allows for optional compression of the data. This can be useful when SSH is used in low-bandwidth situations such as dial-up lines. If the client and server negotiate compression, only the type and data fields are compressed.

Many of these messages either carry no arguments—they consist of only the length, padding, type, and CRC fields—or have a single argument consisting of a string, integer, or extended integer. In these cases, we won't bother showing the message layout but will merely indicate what type of argument, if any, the message carries.

Server Authentication

The server authentication phase of the session, as shown in Figure 7.4, begins with the exchange of identification strings. When the SSH client, `ssh`, connects to the SSH

No.	Message Name	Message
0	SSH_MSG_NONE	never sent
1	SSH_MSG_DISCONNECT	causes immediate connection teardown
2	SSH_SMSG_PUBLIC_KEY	server's public key
3	SSH_CMSG_SESSION_KEY	client choice of cipher and session key
4	SSH_CMSG_USER	user logon name
5	SSH_CMSG_AUTH_RHOSTS	request for user rhosts type authentication
6	SSH_CMG_AUTH_RSA	request for user RSA authentication
7	SSH_SMSG_AUTH_RSA_CHALLENGE	server challenge for RSA authentication
8	SSH_CMSG_AUTH_RESPONSE	client response to RSA challenge
9	SSH_CMSG_AUTH_PASSWORD	request for password authentication
10	SSH_CMSG_REQUEST_PTY	request for a server pseudoterminal
11	SSH_CMSG_WINDOW_SIZE	client's window size
12	SSH_CMSG_EXEC_SHELL	request to start a user shell
13	SSH_CMSG_EXEC_CMD	request to run a command
14	SSH_SMSG_SUCCESS	server accepts last request
15	SSH_SMSG_FAILURE	server does not accept last request
16	SSH_CMSG_STDIN_DATA	client input data for shell/command
17	SSH_SMSG_STDOUT_DATA	output data from shell/command
18	SSH_SMSG_STDERR_DATA	STDERR output from shell/command
19	SSH_CMSG_EOF	client is finished sending data
20	SSH_SMSG_EXITSTATUS	exit status from shell/command
21	SSH_MSG_CHANNEL_OPEN_CONFIRMATION	indicates channel opened
22	SSH_MSG_CHANNEL_OPEN_FAILURE	indicates channel could not be opened
23	SSH_MSG_CHANNEL_DATA	data transmitted over channel
24	SSH_MSG_CHANNEL_CLOSE	sender is closing channel
25	SSH_MSG_CHANNEL_CLOSE_CONFIRMATION	sender acknowledges channel close
26		obsolete
27	SSH_SMSG_X11_OPEN	client is connected to proxy X-server
28	SSH_CMSG_PORT_FORWARD_REQUEST	client requests server port be forwarded
29	SSH_MSG_PORT_OPEN	connection made on forwarded port
30	SSH_CMSG_AGENT_REQUEST_FORWARDING	requests authentication agent forwarding
31	SSH_SMSG_AGENT_OPEN	requests channel to authentication agent
32	SSH_MSG_IGNORE	no op
33	SSH_CMSG_EXIT_CONFIRMATION	response to SSH_SMSG_EXITSTATUS
34	SSH_CMSG_X11_REQUEST_FORWARDING	requests a proxy X-server
35	SSH_CMSG_AUTH_RHOSTS_RSA	requests rhosts/RSA authentication
36	SSH_MSG_DEBUG	debugging information for peer
37	SSH_CMG_REQUEST_COMPRESSION	client requests compression

Figure 7.3 SSHv1 Message Types

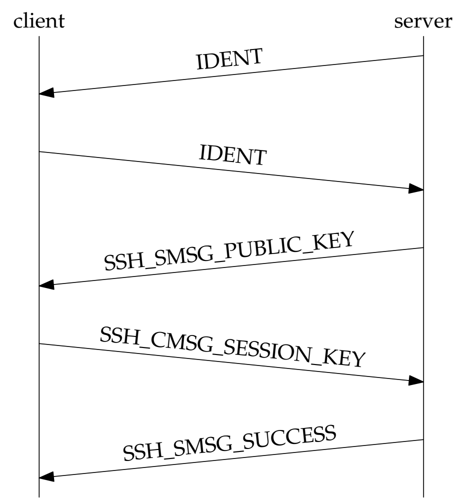

Figure 7.4 SSHv1 Server Authentication

server, `sshd`, the server sends an identification string indicating which of the protocols it supports and, perhaps, additional program version information. For example, if we connect to the SSH server with netcat, the server responds with its identification string:

```
$ nc linux 22
SSH-1.99-OpenSSH_3.5p1
```

The `SSH-1.99` is a special version number that tells the client that the server supports protocol versions 1 and 2. The `-OpenSSH_3.5p1` is meant for human consumption and specifies the version of the OpenSSH SSH server. The client will respond with its own identification string, so that the peers will know which protocol to use.

After the server receives the client's identification string, the peers switch to the binary protocol, and the server sends the SSH_SMSG_PUBLIC_KEY message shown in Figure 7.5.

The *cookie* is 8 random bytes that are intended to make IP spoofing more difficult. The client must return these bytes to the server unchanged.

The *host key* is a permanent RSA public key that the client uses to verify the identity of the server. Unlike with SSL, this key is not signed by a third party. Rather, the client is expected to have a database of known host keys. In practice, this database is built by accepting the key as valid the first time a user connects to a host. Thereafter, the host must present the known key to the client for the session to proceed. SSH also supports a key-fingerprinting mechanism that allows a user to manually verify a site's key with the site's system administrator. There is also a proposal to make these fingerprints available through DNS by means of SSHFP (SSH key fingerprint) records.

The server also sends a second key, the *server key*. This key is regenerated periodically, once every hour by default, to help improve security. As we shall see, the client uses both of these keys to form its response to the server.

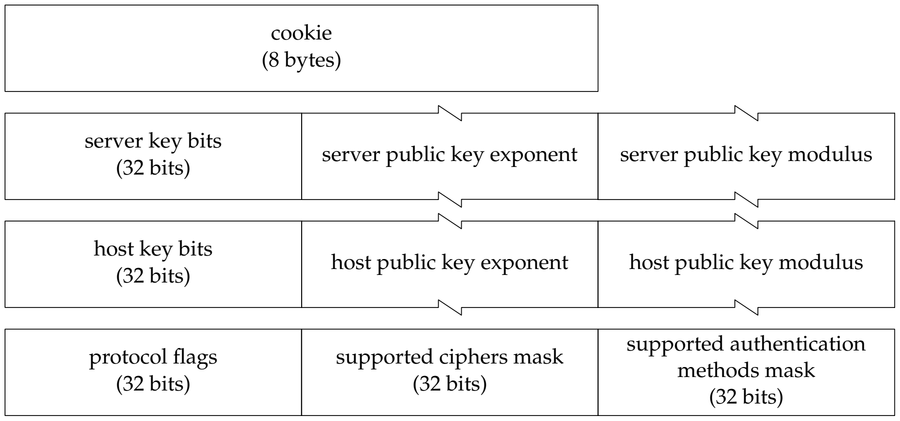

Figure 7.5 SSH_SMSG_PUBLIC_KEY Message

Finally, the message includes three 32-bit bit masks. The *protocol flags* bit mask is intended for protocol extension. The *supported ciphers mask* indicates which ciphers the server can use. The client will choose one of these for the session's cipher. The *supported authentications mask* indicates which user authentications the server supports. Again, the client will try one or more of these methods to authenticate the user.

Both sides use the information in this message to calculate a session ID by taking the MD5 hash of the concatenation of the moduli of the server and host keys and the cookie. As we'll see, the session ID is used in generating the session key and thus ensures that both the client and server contribute to the session key.

From Figure 7.4, we see that the client responds to the SSH_SMSG_PUBLIC_KEY message with an SSH_CMSG_SESSION_KEY message, shown in Figure 7.6.

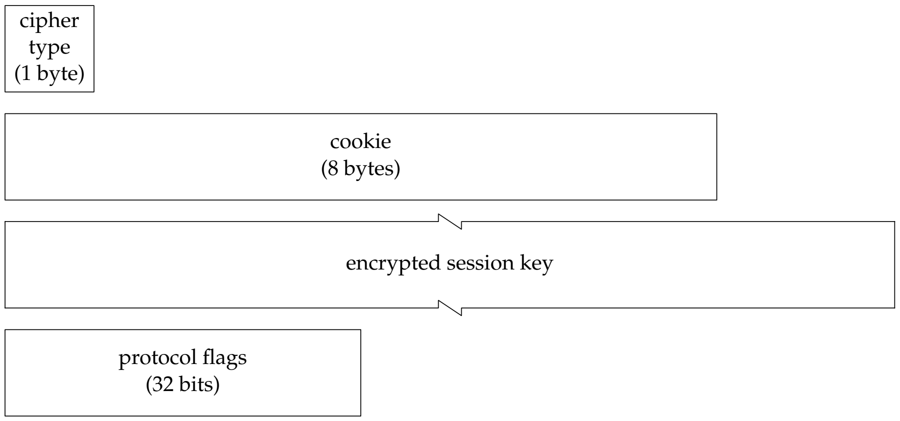

Figure 7.6 SSH_CMSG_SESSION_KEY Message

The *cipher type* field contains the number of the cipher that the client has chosen for the session. The *cookie* and *protocol flags* fields are just as they were for the SSH_SMSG_PUBLIC_KEY message. In particular, the cookie must be returned to the server exactly as it was received by the client.

The *encrypted session key* field is a random 32-byte value chosen by the client. The session ID, calculated from the values in the SSH_SMSG_PUBLIC_KEY message, is exclusive-ORed into the first 16 bytes of the random value. This result is then encrypted twice: first by the smaller (usually the server) RSA key and then by the other (usually host) key.

Although this operation seems complicated, it accomplishes three separate tasks. First, by exclusive-ORing the session ID into the random value used for the session key, the client ensures that both the server and the client contribute to the final session key. Then, by encrypting with the host key, the client verifies the identity of the server, because the server must have the corresponding private key in order to recover the session key. Finally, by encrypting with the server key, the client ensures perfect forward secrecy by using the periodically changing server key.

Both sides now begin encrypting their packets. The server completes the server authentication phase by sending an SSH_SMSG_SUCCESS message. At this point, the peers have established a secure channel, and the server has authenticated itself to the client.

User Authentication

The next step is for the user to authenticate himself to the server. This can be done in several ways. SSH allows, but discourages, the insecure rhosts trusted-host model. Because it is easily spoofed, this model should never be used when security is important. SSH also supports a variation of the rhosts model in which the identity of the client machine is verified with an RSA key. This is an improvement but still relies on the client to certify the identity of the user. Once again, this method should not be used when security or user identity is a concern. The rhosts and rhosts/RSA methods are discussed in detail in [Barrett and Silverman 2002], so we will not belabor them further here.

A third authentication method is to use Kerberos. With this method the user obtains a "ticket" from the Kerberos server and sends it to the SSH server as authentication. Although Kerberos is a complicated system and requires a separate server, it may make sense when there is a large user base, especially if Kerberos is already in place. See [Garman 2003] for more information on Kerberos.

Next, there is a class of methods known as password authentication. In the simplest of these, which we'll examine shortly, the user merely supplies a password, which the server checks against its password file. Recall that after server authentication, the peers have established a secure channel, so this password is not passed in the clear as it is in, say, the telnet protocol.

The other password methods are variations of a one-time-password scheme. One example is the popular RSA SecurID system, which is described at RSA's Web site (<`http://www.rsasecurity.com/node.asp?id=1156`>). With SecurID, the user carries a hardware device, such as a key fob, that generates a pseudorandom number every 60 seconds. During authentication, the user enters the pseudorandom number and a private PIN. If the correct values are entered, the user is authenticated, and the session is started.

Another one-time-password scheme is the S/Key system [Haller 1994, Haller 1995], a challenge/response system. With S/Key, the user is prompted with a challenge and responds with a set of short words. This response can be obtained either programmatically—from a PDA applet, say—or from a preprinted list. As with SecureID, passwords are not reused, thereby increasing security. The S/Key system is supported by OpenSSH.

Each of these password methods has its own protocol, but they all depend on the SSH secure channel for their security. The enhanced methods, such as SecureID and S/Key, derive their increased security from the fact that they use one-time passwords and are thus resistant to local passive attacks, such as keyboard loggers and other attempts to capture the password before it enters the encrypted channel; even if a password is captured, it is useless because it's not reused.

Figure 7.7 shows the protocol for a simple password-based user authentication.

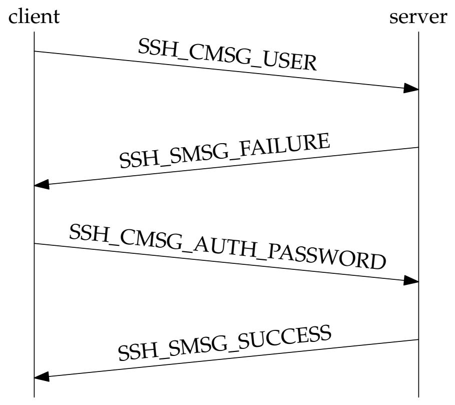

Figure 7.7 User Authentication with a Password

First, the client sends the server the user's name as a string in the SSH_CMSG_USER message. If no further authentication is required—rhosts authentication is being used, for example—the server will respond with an SSH_SMSG_SUCCESS message, and authentication will be completed. If further credentials, such as a password, are required, the server will respond with an SSH_SMSG_FAILURE message, indicating that the user name alone is not sufficient. SSH_SMSG_SUCCESS and SSH_SMSG_FAILURE are both simple messages with no arguments.

At this point, the client will try various authentication methods until it finds one acceptable to the server. In our example session, the client first tries a standard password authentication by sending the server an SSH_CMSG_AUTH_PASSWORD message containing the user's password as a string. In our case, this is acceptable, and the server returns an SSH_SMSG_SUCCESS message indicating that the authentication is complete.

Because SSHv1 normally encrypts everything in its messages except the length, there is not much sense in dwelling on packet captures as a debugging aid. Nevertheless, it is instructive to see what the messages look like, to aid us in understanding the protocol. Therefore, let's briefly examine the messages from Figure 7.7 in detail.

We begin with the SSH_CMSG_USER message from the client. As we see from bytes 5–8 in line 1.4, the message is 14 bytes long. From line 1, we see that the TCP segment is 20 bytes long, so there are 20 – 14 – 4 = 2 bytes of padding. The padding is easy to spot because OpenSSH pads with zero bytes.

Next, we see the message type (0x04) in byte 11 of line 1.4. From Figure 7.3, we see that this is indeed the SSH_CMSG_USER message. Following the message type is the user's logon name, guest in this case. In SSH, string data is preceded by 4 bytes of length followed by the string data. There is no trailing NULL byte. Finally, the last 4 bytes are the CRC, which serves as a MAC.

In line 2, the server responds with its SSH_SMSG_FAILURE message. The message type (0x0f) is on line 2.4. Line 3 is the TCP ACK for the segment in line 2. This occurs because of the time it takes to type in the password.

```
1    14:56:52.381285 127.0.0.1.32802 > 127.0.0.1.2022: P 184:204(20)
       ack 316 win 3276 7 <nop,nop,timestamp 468782 468782> (DF)
1.1      4500 0048 0daa 4000 4006 2f04 7f00 0001    E..H..@.@./.....
1.2      7f00 0001 8022 07e6 2b99 26b8 2b24 4d08    ....."..+.&.+$M.
1.3      8018 7fff d4cb 0000 0101 080a 0007 272e    ..............'.
1.4      0007 272e 0000 000e 0000 0400 0000 0567    ..'...........g
1.5      7565 7374 33b3 5ce1                        uest3..
2    14:56:52.381867 127.0.0.1.2022 > 127.0.0.1.32802: P 316:328(12)
       ack 204 win 3276 7 <nop,nop,timestamp 468782 468782> (DF)
2.1      4500 0040 0dab 4000 4006 2f0b 7f00 0001    E..@..@.@./.....
2.2      7f00 0001 07e6 8022 2b24 4d08 2b99 26cc    ......."+$M.+.&.
2.3      8018 7fff a93e 0000 0101 080a 0007 272e    .....>........'.
2.4      0007 272e 0000 0005 0000 000f 90bf 1d91    ..'.............
3    14:56:52.413821 127.0.0.1.32802 > 127.0.0.1.2022: . ack 328
       win 32767 <nop,nop,timestamp 468786 468782> (DF)
```

Next, we see the SSH_CMSG_AUTH_PASSWORD message (type 0x09) carrying the password of knockknock. Because this password is acceptable to the server, it responds with a SSH_SMSG_SUCCESS message in line 5. As we see in line 5.4, the SSH_SMG_SUCCESS message carries no data other than its message type (0x0e).

```
4    14:56:59.534978 127.0.0.1.32802 > 127.0.0.1.2022: P 204:256(52)
             ack 328 win 3276 7 <nop,nop,timestamp 469498 468782> (DF)
4.1      4500 0068 0dba 4000 4006 2ed4 7f00 0001    E..h..@.@.......
4.2      7f00 0001 8022 07e6 2b99 26cc 2b24 4d14    ....."..+.&.+$M.
4.3      8018 7fff ecd7 0000 0101 080a 0007 29fa    ..............).
4.4      0007 272e 0000 0029 0000 0000 0000 0009    ..'....).......
4.5      0000 0020 6b6e 6f63 6b6b 6e6f 636b 0000    ....knockknock..
4.6      0000 0000 0000 0000 0000 0000 0000 0000    ................
4.7      0000 0000 7ba8 d3b8                        ....{...
5    14:56:59.536726 127.0.0.1.2022 > 127.0.0.1.32802: P 328:340(12)
             ack 256 win 3276 7 <nop,nop,timestamp 469498 469498> (DF)
```

```
5.1     4500 0040 0dbc 4000 4006 2efa 7f00 0001    E..@..@.@.......
5.2     7f00 0001 07e6 8022 2b24 4d14 2b99 2700    ......."+$M.+.'.
5.3     8018 7fff 3cf8 0000 0101 080a 0007 29fa    ....<.........).
5.4     0007 29fa 0000 0005 0000 000e e7b8 2d07    ..)...........-.
```

The final type of user authentication, RSA authentication, uses RSA keys as certificates. The public key is placed in the user's home directory on the server, usually in the .ssh subdirectory. Similarly, the private key is kept in the user's home directory on the client. If the client has an RSA key available, it will query the server as to whether it has the corresponding public key. It does this by sending the key's modulus in an SSH_CMSG_AUTH_RSA message. The entire RSA authentication process is shown in Figure 7.8.

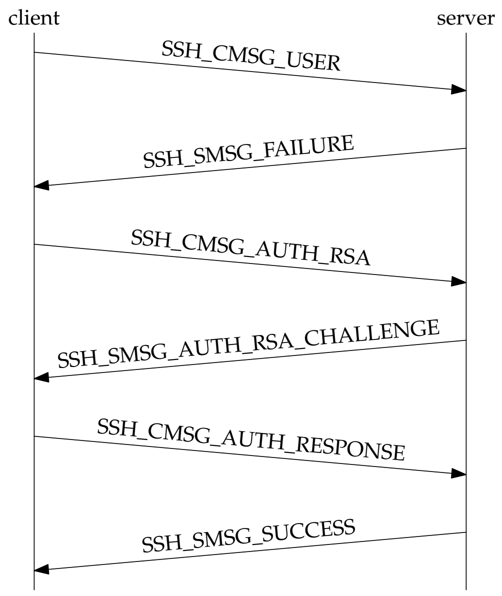

Figure 7.8 SSHv1 RSA Authentication

The exchange starts with the client sending the usual SSH_CMSG_USER message with the user's logon name. The server responds with the SSH_SMSG_FAILURE message, indicating that further authentication is required. So far, this is the same as for password authentication, but with RSA authentication instead of the client sending the SSH_CMSG_AUTH_PASSWORD message, it sends an SSH_CMSG_AUTH_RSA message containing the modulus of the RSA key as an extended integer. The server checks in the user's home directory for a public key with the same modulus. If it finds one, the server uses it to encrypt a random 256-bit challenge, which it sends to the client in an SSH_SMSG_AUTH_RSA_CHALLENGE message. The challenge is carried as an extended integer. The client decrypts the challenge with its private key, combines it with the session ID, and takes an MD5 hash of the result. The hash is sent to the server in an SSH_CMSG_AUTH_RESPONSE message as 16 bytes of untyped data. The server takes its own MD5 hash of the challenge and session ID and compares it with the one it received from the client. If they match, the server sends the client an SSH_SMSG_SUCCESS message, indicating that the authentication succeeded. If the results don't match, the server will respond with an SSH_SMSG_FAILURE message, informing the client that it should try another form of user authentication.

Once user authentication is completed, SSH is ready to begin a user application. By default, this means that SSH will start a user shell on the server and transport the shell's standard input, output, and error over the encrypted channel. We show this for the case of our sample session in Figure 7.9.

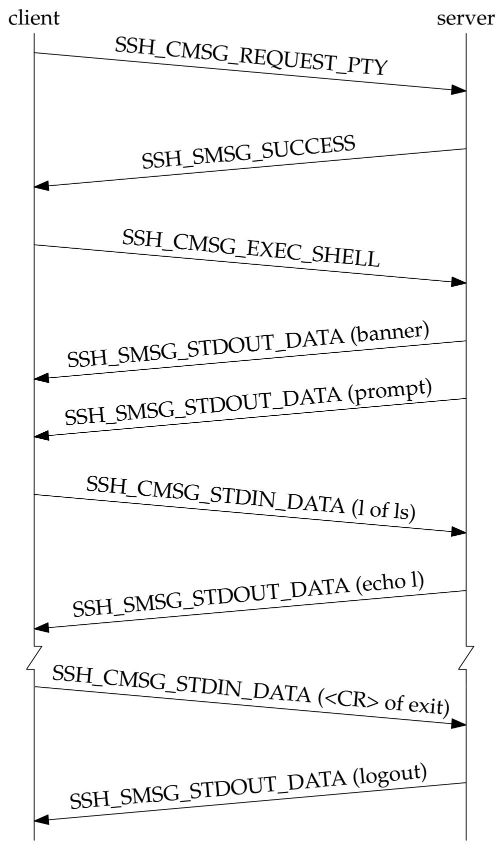
client
server
SSH_CMSG_REQUEST_PTY
SSH_SMSG_SUCCESS
SSH_CMSG_EXEC_SHELL
SSH_SMSG_STDOUT_DATA (banner)
SSH_SMSG_STDOUT_DATA (prompt)
SSH_CMSG_STDIN_DATA (l of ls)
SSH_SMSG_STDOUT_DATA (echo l)
SSH_CMSG_STDIN_DATA (<CR> of exit)
SSH_SMSG_STDOUT_DATA (logout)

Figure 7.9 Starting a User Shell

First, the client requests the server to start a pseudo-tty session for it with the SSH_CMSG_REQUEST_PTY message. This message contains the terminal type and characteristics of the user's session, as shown in Figure 7.10. After the server acknowledges the request with an SSH_SMSG_SUCCESS message, the client requests the server to start a shell and connect it to the pseudo-tty with an SSH_CMSG_EXEC_SHELL message. The client and server then exchange data with the SSH_CMSG_STDIN_DATA and SSH_SMSG_STDOUT_DATA messages. The SSH_CMSG_EXEC_SHELL message contains no arguments. The arguments for the SSH_CMSG_STDIN_DATA and SSH_SMSG_STDOUT_DATA messages are carried as string data.

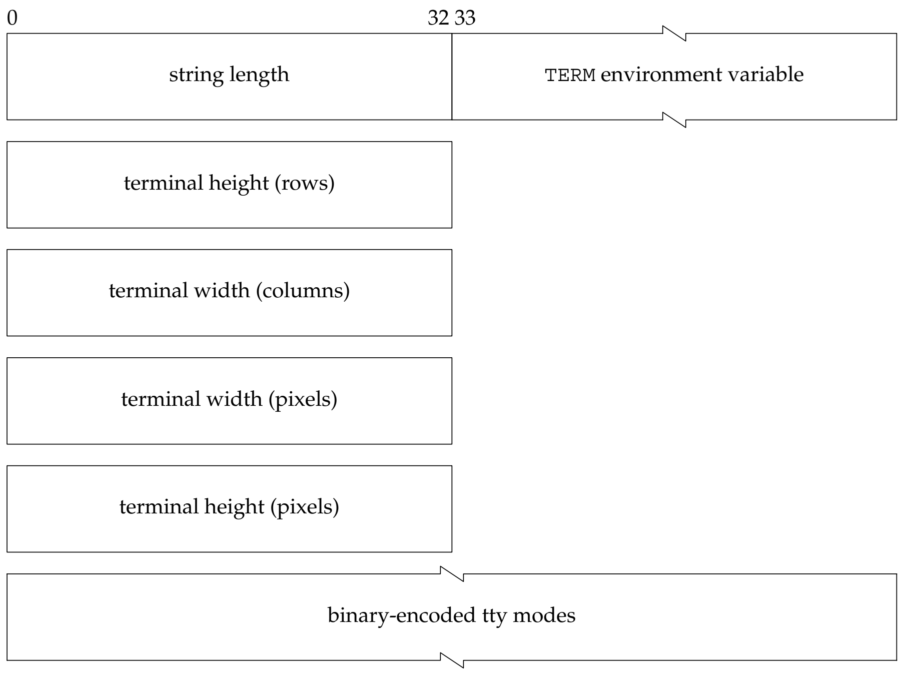

Figure 7.10 The SSH_CMSG_REQUEST_PTY Message

When we typed `exit`, the shell sent a final ''logout'' message and terminated. That caused SSH to tear down the encrypted channel, as shown in Figure 7.11.

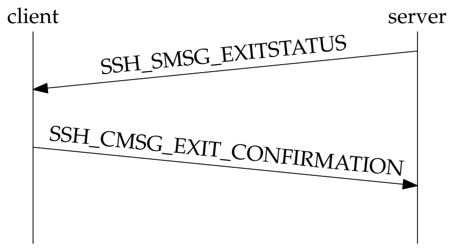

Figure 7.11 Channel Teardown

First, the server sent an SSH_SMSG_EXITSTATUS message containing the exit status of the shell. If the shell had been terminated by a signal, the server would have sent an SSH_MSG_DISCONNECT message containing a human-readable string indicating the cause of the termination. The client confirms the shell termination by responding with a SSH_CMSG_EXIT_CONFIRMATION message, which contains no arguments. These two messages are the last messages sent by the server and the client, respectively.

X11 Forwarding

Both versions of SSH can forward an X11 connection from an X-client on the SSH server to an X-server on the SSH client. This works by having the SSH server listen for connections to the X-server from local—to the server—X-clients, in effect establishing an X11-server proxy. The SSH server then forwards the X-protocol messages from the X-client over an encrypted channel to the SSH client, which in turn forwards them to the real X-server on the SSH client side. Similarly, X-protocol messages from the X-server on the client are transferred over the encrypted channel to the X-client on the SSH server. In Figure 7.12, an X-client on the server host connects to the X-server proxy and has its X-protocol messages delivered over the SSH encrypted channel to the X-client proxy on the client host. The X-client proxy in turn connects to the real X-server and forwards messages from the X-client to it. Although both the X-client and X-server believe themselves to be directly connected to their peer and that their peer is on the same machine, the actual connection follows the shaded path shown in Figure 7.12.

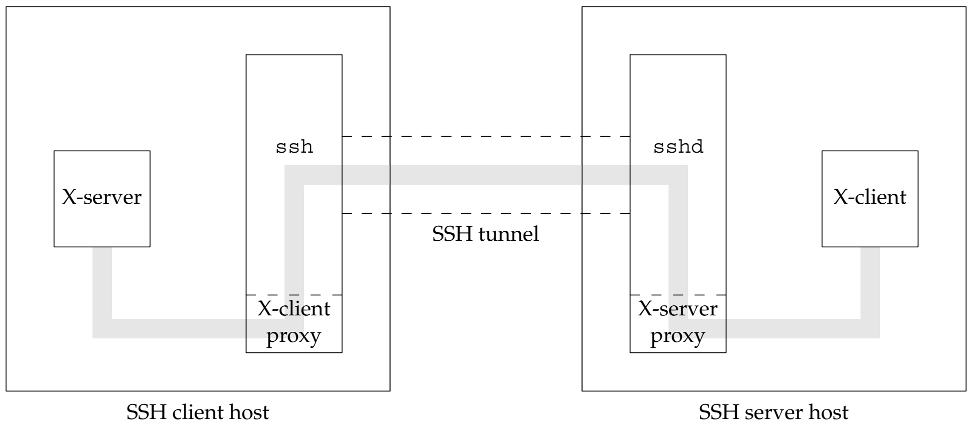

Figure 7.12 A Forwarded X11 Connection

If the SSH client wishes to enable X11 forwarding, it sends the SSH server an SSH_CMSG_X11_REQUEST_FORWARDING message after completing the user authentication phase. This message will cause the server to arrange for local X-clients to connect to a socket on which it is listening.

The server does this by pointing the environment's `DISPLAY` variable at a socket on which it listens for connections. For example, if we `ssh` into `linuxlt` from `bsd`, specifying X11 forwarding, and echo the `DISPLAY` variable, we get

```
echo $DISPLAY
localhost:10.0
```

This corresponds to listening on the loopback interface at port 6010. If we echo the `DISPLAY` variable from the console of `linuxlt`, we get

```
echo $DISPLAY
:0
```

which corresponds to listening on all interfaces at port 6000. We can verify this by running `netstat -naAinet` on `linuxlt`. The relevant entries, after some reformatting, are as expected:

```
Proto  Local Address    Foreign Address  State
tcp    0.0.0.0:6000     0.0.0.0:*        LISTEN
tcp    127.0.0.1:6010   0.0.0.0:*        LISTEN
```

The SSH_CMSG_X11_REQUEST_FORWARDING message is shown in Figure 7.13.

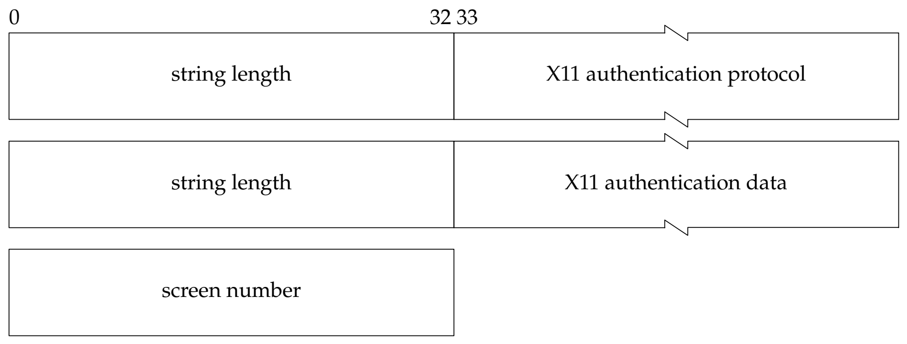

Figure 7.13 The SSH_CMSG_X11_REQUEST_FORWARDING Message

The *authentication protocol* and *authentication data* fields contain data that the proxy X-server can use to authenticate X-clients on the SSH server. See [Barrett and Silverman 2002] for details on X11 authentication and authentication spoofing. The *screen number* field specifies the screen number to use. This field is present only if the SSH_PROTOFLAG_SCREEN_NUMBER bit is set in the protocol flags field of the client's SSH_CMSG_SESSION_KEY message.

When an X-client connects to the proxy X-server, the SSH server sends an SSH_SMSG_X11_OPEN message to the SSH client. This message contains a 32-bit integer field specifying the channel number that it will use for the connection and an optional—if both sides set the SSH_PROTOFLAG_HOST_IN_FWD_OPEN bit in their protocol flag fields during key exchange—string containing a description of the host opening the X11 connection.

The client responds by trying to connect to the real X11 server and sending an SSH_MSG_CHANNEL_OPEN_CONFIRMATION if the connection to the X-server succeeded or an SSH_MSG_CHANNEL_OPEN_FAILURE message if the connection failed.

The SSH_MSG_CHANNEL_OPEN_CONFIRMATION message, shown in Figure 7.14, contains two 32-bit integers. The *remote channel* field is the peer's identifier for the channel over which the X-protocol will be sent. It is the value contained in the SSH_SMSG_X11_OPEN message. The *local channel* field is the SSH client's identifier for the secure channel.

This is reminiscent of L2TP, in which each of the peers had its own identifier for a data channel.

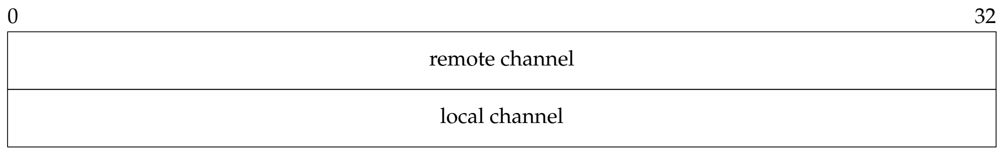

Figure 7.14 The SSH_MSG_CHANNEL_OPEN_CONFIRMATION Message

The SSH_MSG_CHANNEL_OPEN_FAILURE message contains a single 32-bit integer that is the peer's identifier for the channel. In the context of X11 forwarding, this message means that the SSH client was unable to connect to the local X-server.

Once the client sends the SSH_MSG_CHANNEL_OPEN_CONFIRMATION, a secure, bidirectional data channel is available to the peers. As noted previously, the SSH server will use it to forward X-protocol messages from a local X-client, and the SSH client will use it to forward X-protocol messages from the X-server back to the X-client on the SSH server. The peers use the SSH_MSG_CHANNEL_DATA message, shown in Figure 7.15, to carry the channel data. The *remote channel* field is a 32-bit integer containing the *peer's* identifier for the channel. The data is carried as a string in the *data* field.

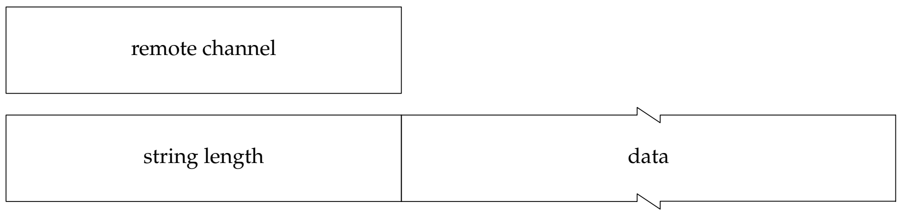

Figure 7.15 The SSH_MSG_CHANNEL_DATA Message

Either side of the X11 connection may close it by sending the SSH_MSG_CHANNEL_CLOSE message, which contains the peer's value for the channel number as an integer. For example, the SSH server will send this message when the X-client disconnects from the X-proxy server. The other side will let any queued data drain from the connection and will then send an SSH_MSG_CHANNEL_CLOSE_CONFIRMATION message to complete the channel teardown. Figure 7.16 shows the SSH protocol exchange in a typical forwarded X11 session.

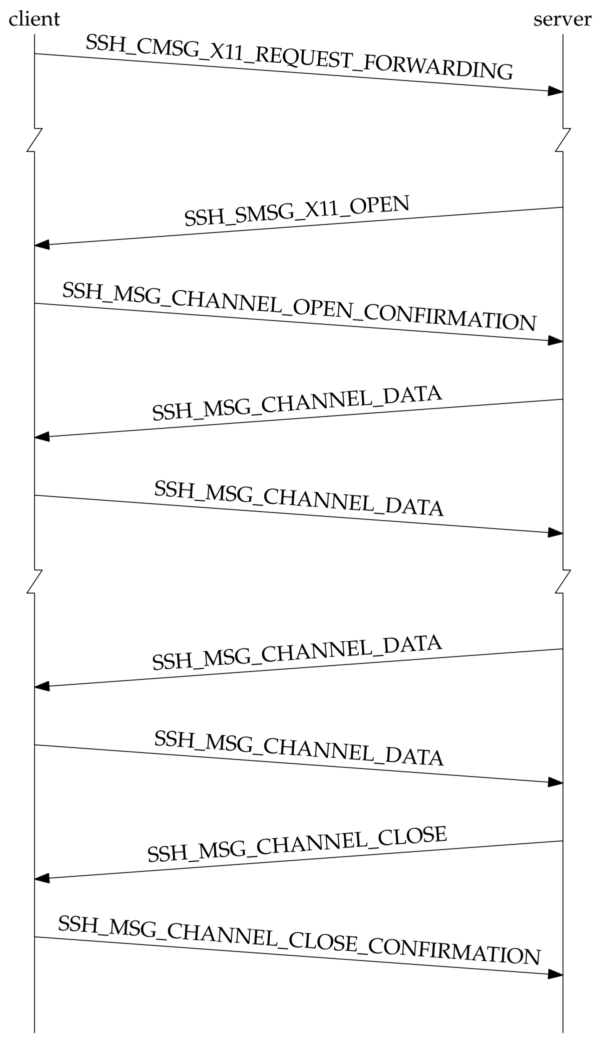

Figure 7.16 Forwarded X11 Connection Flow

Port Forwarding

In the previous subsection, we examined SSH X11 forwarding. Both SSH versions also have a more general connection forwarding capability called *port forwarding*. This capability is much like that provided by `stunnel`. The most general case is shown in Figure 7.17: host A wants to connect securely to port R on host B. Instead of directly connecting—possibly across the Internet or other insecure path—to host B, host A connects to port L on the SSH client host instead. SSH then forwards this connection through a secure channel to the SSH server that connects to port R on host B. Thereafter, data from host A travels to the SSH client on the local connection, across the SSH tunnel, and finally across the local connection that the SSH server made to host B. Host B is unaware that forwarding is going on and merely listens for connections on port R as usual. Host A connects to the SSH client, which is listening on port L, instead of to host B.

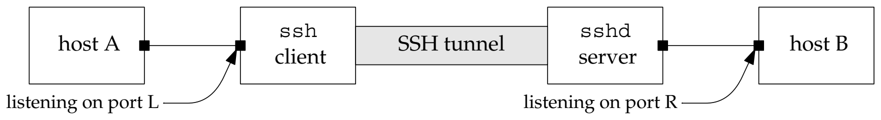

Figure 7.17 SSH Local Port Forwarding

For security reasons, the default behavior of the SSH client is to listen for local connections only from applications on the same machine. A special flag to `ssh` is required to get the behavior shown in Figure 7.17. Often the remote application will be on the same machine as the SSH server, but the protocol makes no assumptions about this.

It is also possible to have remote applications connect back to the SSH client host or to a host reachable by the SSH client, as shown in Figure 7.18.

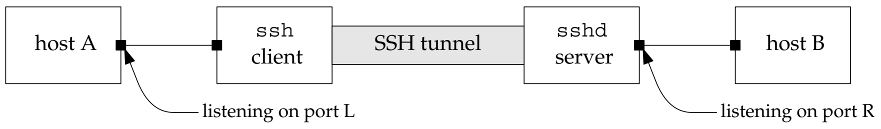

Figure 7.18 SSH Remote Port Forwarding

In this case, host B wants to connect to host A on port L. When the SSH client requests remote port forwarding, the SSH server listens for connections on the specified port: say, port R. When an application connects to port R, the connection is forwarded through the SSH tunnel to the SSH client, which then connects to port L on the target host: host A in Figure 7.18.

As we mentioned in Chapter 5 during our discussion of L2TP/IPsec, it is important to understand the *end-to-end* security of the tunnels that we set up. For example, if the client application and server application that we port forward between are on the same machines as the SSH client and server, the entire path between application and corresponding server is protected. If, on the other hand, we have a situation such as that in Figure 7.17 or Figure 7.18, parts of the path are unprotected. This may or may not be a problem. If, for instance, host A of Figure 7.17 is on the same trusted network as the SSH client machine, the fact that the connection between host A and the SSH client is not secured may be okay. The same configuration may not be okay if the connection between host A and the SSH client machine is exposed to untrusted parties. Thus, it is important to understand both the threat model and the security of each leg of a proposed VPN.

Let's look at the SSH protocol messages involved in forwarding a local port. We assume that the user has requested SSH to forward port L on the SSH client to port R on a remote server reachable by the SSH server and that the server is the first to disconnect, initiating the channel teardown. Figure 7.19 shows the SSH message flow.

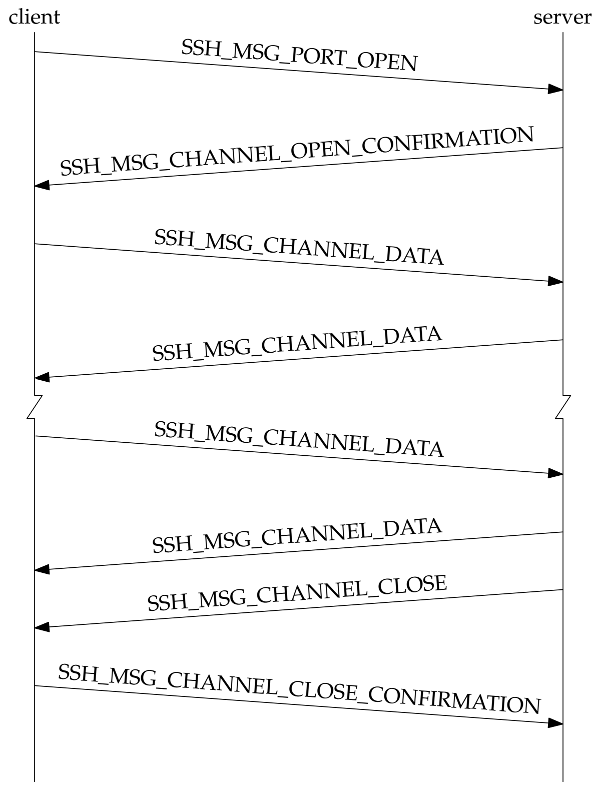

Figure 7.19 Port Forwarding Protocol Exchange

The process begins when an application connects to port L on the SSH client host. The SSH client responds to the connection by sending the SSH server an SSH_MSG_PORT_OPEN message indicating that a connection has been made to a port for which the user requested forwarding and asking the server to connect to the specified host and port. The SSH_MSG_PORT_OPEN message is shown in Figure 7.20. The *local channel* field is the client's identification number for the channel. The SSH server will use this channel number when sending data to or about this connection to the client.

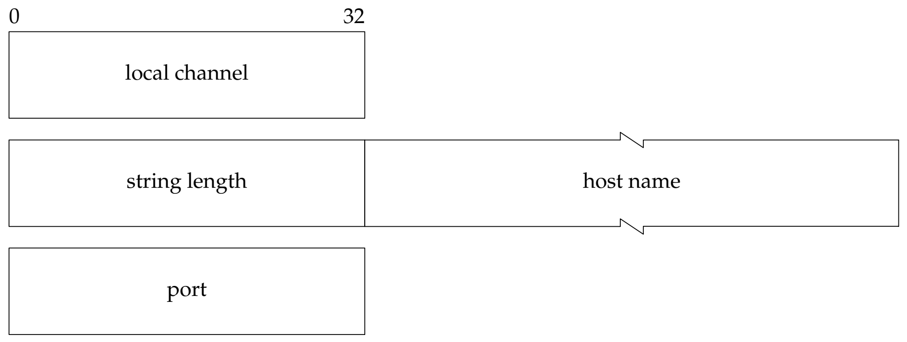

Figure 7.20 The SSH_MSG_PORT_OPEN Message

The *host name* is the name or address of the host that the SSH server should connect to. Similarly, the *port* field is the port to which it should connect.

After connecting to the specified host, the SSH server sends an SSH_MSG_CHANNEL_OPEN_CONFIRMATION message. If it can't connect to the remote host, the SSH server will send an SSH_MSG_CHANNEL_OPEN_FAILURE message instead. Assuming that the connection succeeds, the two (non-SSH) peers can start exchanging data, which is carried in SSH_MSG_CHANNEL_DATA messages, just as in X11 forwarding.

If the remote host disconnects from the SSH server, as shown in Figure 7.19, the SSH server will send an SSH_MSG_CHANNEL_CLOSE message, and the SSH client will respond with an SSH_MSG_CHANNEL_CLOSE_CONFIRMATION message after shutting down its end of the connection. If the local host disconnects first, the actions are the same except that the SSH client sends the SSH_MSG_CHANNEL_CLOSE, and the SSH server responds.

The protocol exchange for remote port forwarding is the same except that the SSH client must first tell the SSH server to start listening on a port—port R, say—for a connection from the remote host. This is accomplished by having the SSH client send an SSH_CMSG_PORT_FORWARD_REQUEST message, shown in Figure 7.21, right after user authentication. The *server port* field is the port on which the server is to listen—port R in our example. The *host name* and *port* fields are the name or address and port to which the connection should be forwarded. These are the values that will be sent in the host name and port fields of the SSH_MSG_PORT_OPEN message when the remote host connects.

Running Remote Programs Through an SSH Tunnel

As mentioned previously, the most common use of SSH is to run a shell on a remote host. We can, however, ask SSH to start any program we please on the remote host and send its STDIN, STDOUT, and STDERR through the SSH tunnel.

Suppose, for example, that we want to run our `echoit` program on `bsd` securely from `linuxlt`:

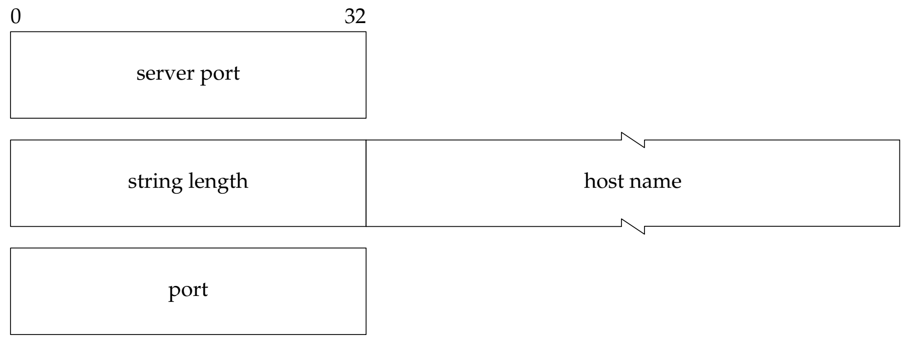

Figure 7.21 The SSH_CMSG_PORT_FORWARD_REQUEST Message

```
$ ssh -t bsd echoit
Hello, world!
Hello, world!
^D                                          send an EOF to echoit
Connection to bsd closed.
```

The -t tells SSH to start a pseudo-tty for us. Because we are dealing with the remote program interactively through a terminal session, we need the pseudo-tty to handle terminal control characters and similar events for us.

Note that we don't get the usual login banner or shell prompt. That's because the SSH server executes our program directly, rather than starting a shell and letting us execute it from the shell.

> Actually, even here the shell is involved. The SSH server executes the command by calling the shell with the -c switch. That is, the SSH server runs our command as
>
> ```
> /usr/local/bin/bash -c echoit
> ```
>
> where the /usr/local/bin/bash was obtained from the SHELL environment variable on bsd.

Also note that when we send echoit an EOF, its termination causes the SSH connection to be torn down just as if we had logged out of a shell connection.

From a protocol standpoint, running a remote program instead of a shell uses the same exchange of protocol messages that we saw in Figure 7.9, except that the SSH_CMSG_EXEC_SHELL message is replaced by an SSH_CMSG_EXEC_CMD message that contains the command to be run as a string.

The Secure Copy Program (SCP)

Using SSH to execute a trivial program, such as echoit, isn't very exciting, of course, and it doesn't begin to illustrate the power of the capability. When Tatu Ylönen wrote SSH, he intended it to replace the rcp routine, which copies files between local and remote hosts, as well as rsh and the other r-commands. To this end, he wrote a separate program, scp, to replace rcp.

The `scp` program is really just a patched version of `rcp` that sends it data through an SSH tunnel.

To understand how this works, let's suppose that we have two UNIX hosts: a local host, `LH`, and a remote host, `RH`. An instance of the SSH server, `sshd`, is listening for connections on the remote host, but otherwise no programs relevant to our discussion are running. On the local host, a user wishing to transfer the file `main.c` to the remote host invokes `scp` as

```
scp main.c RH:
```

To transfer the file securely, `scp` needs two things:

1. A secure tunnel between the local and remote hosts to send the data through
2. Another copy of `scp` running on the remote host to accept and store the file

The `scp` program can arrange for both of these things by spawning, via `fork` and `exec`, an instance of the SSH client as a child process and requesting it to execute a remote instance of `scp`. With OpenSSH, `scp` would invoke `ssh` as

```
ssh -x -oForwardAgent no -oClearAllForwardings yes RH scp -t main.c
```

The flags to `ssh` disable X11, port, and agent forwarding because this invocation of SSH will be used only to transfer the file.

See [Barrett and Silverman 2002] for information about agent forwarding.

Notice that `ssh` is asked to execute the following command on the remote host:

```
scp -t main.c
```

The `-t` flag tells the remote `scp` instance that it will be receiving a file, which it should store as `main.c`.

The flags to `scp` vary, depending on whether a single file is being transferred *to* or *from* the remote host or whether one or more files are being transferred to a directory on the remote host.

The four processes involved are shown in Figure 7.22. The dashed boxes represent child processes. The arrows between the `scp` and SSH instances represent pipes that connect the parent process to the STDIN, STDOUT, and STDERR of the child process.

At this point, the two `scp` instances are connected through the SSH tunnel, and they perform the normal `rcp` protocol exchange to transfer the file. The important point here is that SSH has no special support for file transfers; it merely executes a remote application in the normal manner. Other distributed applications can be implemented in the same way.

As an example of how easy it is to implement secure distributed applications in this way, let's imagine that we have our mail delivered to a central server—`RH`, say—that for security reasons is reachable only by SSH. We want to know when new email has arrived without having to `ssh` onto the server to check. We can do this with a set of

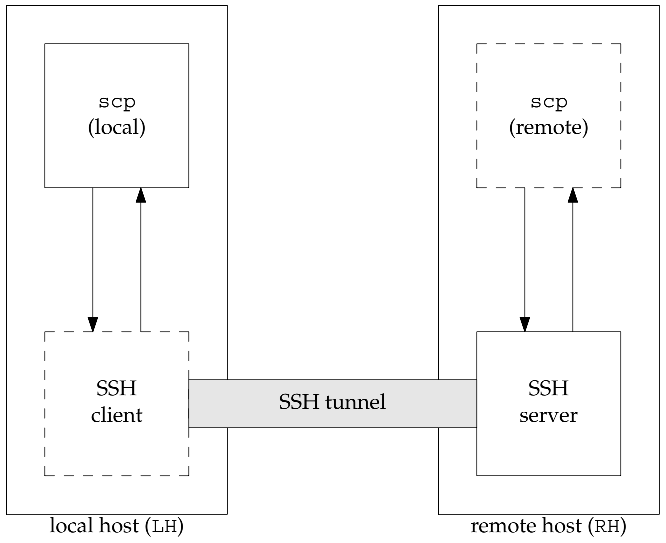

Figure 7.22 An `scp` Connection

`python` programs: `rbiff` and `rbiffd`. We start `rbiff` on our local workstation and it periodically invokes `rbiffd` on the server to check for mail.

Figure 7.23 shows the `rbiff` program.

rbiff

```
 1 #! /usr/bin/env python

 2 import os
 3 from time import *

 4 def notify():
 5     print 'New Mail\n'

 6 while True:
 7     f = os.popen( "ssh RH ./rbiffd", "r" )
 8     for line in f:
 9         if line.startswith( 'New' ):
10             notify()
11             break
12     f.close()
13     sleep( 600 )
```

rbiff

Figure 7.23 The `rbiff` Application

2–3 These lines import two modules that provide an interface to the system commands and time routines. We need these modules for the `popen` and `sleep` functions.

4–5 The `notify` function is called to provide notification when new mail arrives. For simplicity, we show it as just printing "New Mail," but an actual application would pop up a dialog box or change an icon.

6 This is the event loop of the application. Every 10 minutes (see line 13), this loop checks for mail on the server.

7 This is the only part of the two programs that is aware of or deals with SSH. We open a pipe to an instance of `ssh` that executes `rbiffd` on the server `RH`.

8–13 We read from the pipe, looking for a line that begins with "New." If we find one, we call `notify` to alert the user. When we get an EOF from the pipe, we close the pipe and sleep for 10 minutes.

The `rbiffd` (Figure 7.24) program has no knowledge of SSH. When invoked, the program merely checks the mailbox and writes its finding to STDOUT. We could run it directly from the command line without any change at all.

rbiffd
```
1 #! /usr/bin/env python
2 import os

3 mailbox = os.environ[ 'MAIL' ]
4 if newmail( mailbox ):
5     print 'New Mail'
6 else:
7     print 'No New Mail'
```
rbiffd

Figure 7.24 The `rbiffd` Application

2–3 We import the `os` module so that we can get the location of the user's mailbox from the environment.

4–7 We call the `newmail` function, not shown, to check whether new mail is in the user's mailbox. If so, "New Mail" is output to STDOUT. Otherwise, "No New Mail" is output.

We've omitted the `newmail` function to avoid getting bogged down in the minutia of checking for new mail, but the process involves merely checking the times that the mailbox was last written to and accessed.

Security Issues with SSHv1

As with any other program, attackers and security researchers have discovered bugs, such as potential buffer overflows, in the various implementations of SSHv1 that lead to exploits. These bugs were, of course, fixed as they were discovered, so they aren't a concern to us except, perhaps, as a cautionary tale on how difficult it is to implement software—especially security software—in a robust and secure manner. On the other hand, we *do* need to be concerned with bugs in the protocol itself. These bugs are more difficult to deal with because fixing them requires fixing the protocol rather than merely patching a particular implementation.

One example of such a vulnerability is the session key recovery exploit based on Bleichenbacher's attack on RSA messages encoded with PKCS #1 v1.5 [Bleichenbacher 1998]. This exploit is a result of problems within the SSHv1 protocol, not of any particular implementation of the protocol. The basic idea is that the attacker captures the SSH_CMSG_SESSION_KEY message and then uses a combination of a timing attack and Bleichenbacher's attack to decode the SSH_CMSG_SESSION_KEY message and recover the session key. Once the session key is known, of course, the entire (captured) message stream can be read with ease.

Bleichenbacher's attack is an adaptive procedure that sends the SSH server carefully chosen ciphertext messages and observes the behavior of the server. From these observations, the attacker is able to repeatedly narrow the interval that the session key lies in. The attack, which is described in some detail in [Waissbein and Friedman 2001], takes about $2^{20} + 2^{19}$ messages to the server. Because the server key expires every hour, the attacker must make about 400 connections per second to the server.

Many servers can't support that rate of connection, and others, such as OpenSSH, limit the number of concurrent unauthenticated connections, so not all servers are vulnerable. Note, however, that this invulnerability is serendipitous, not the result of these implementations being better coded than those that are vulnerable. The problem here is that the SSHv1 protocol causes SSHv1 servers to leak information and thus act as "oracles" for the Bleichenbacher attack.

Fortunately, vulnerable servers can be patched to prevent this exploit by regenerating the server key when they detect the attack. The longer-term solution was to change the PKCS #1 encoding scheme to make it resistant to this type of attack. See [Bleichenbacher, Kalisky, and Staddon 1998] for a description of Bleichenbacher's attack and the changes made to PKCS #1 to prevent it.

Another, more serious, problem with the SSHv1 protocol was not so easily fixed and resulted in the abandonment of SSHv1 and the development of SSHv2. Recall that SSHv1 uses a CRC instead of a MAC. Although quite good at detecting random errors introduced into a file, CRCs make weak MACs. A MAC based on a cryptographic hash, such as MD5 or SHA-1, acts as a pseudorandom function.

> These functions aren't truly random, of course, because they are deterministic. The point is that the output *appears* not to depend on the input in any predictable way. In particular, pseudorandomness requires that given part of the output of the function but not the input, it is infeasible to determine the rest of the output.

Put another way, it's extremely difficult to construct a message m having a particular hash. CRCs, on the other hand, have relatively simple mathematical properties that make it easier to construct a message m with a given CRC value.

An exploit based on the weakness of CRC-32 as a MAC allows the insertion of arbitrary data into an SSHv1 message stream. That is, the attacker is able to insert text into the encrypted message stream, which then decrypts to text of the attacker's choice. This means, among other things, that an attacker can execute arbitrary commands on the SSH server or can display arbitrary data on the client's screen. The details of the attack are given in [Futoransky, Kargieman, and Pacetti 1998].

This exploit was much more difficult to fix than the Bleichenbacher attack. SSHv1 servers were patched with the so-called *CRC-32 compensation attack detector*, but this merely made the attack more difficult, not impossible. At this time, most security experts recommend against using SSHv1 in any environment where active attacks are a serious concern.

7.3 The SSHv2 Protocol

In this section, we explore the version 2 SSH protocol. SSHv2 adds features and deals with security problems in SSHv1, but the design is considerably different and more flexible.

At the time of this writing, SSHv2 is still in the RFC draft stage, so in one sense the protocol is not completely defined. On the other hand, SSHv2 is widely deployed by several vendors, so few substantive changes are likely to the base protocol as it now exists. The latest draft documents are available from the IETF SECSH working group at its Web site.

From an architectural standpoint, SSHv2 replaces the monolithic protocol of SSHv1 with a layered set of protocols, as shown in Figure 7.25. The SSHv2 transport protocol uses TCP to carry its messages. The SSHv2 authentication and connection protocols, in turn, use the SSHv2 transport protocol to carry their messages. We will look at each of these protocols in detail.

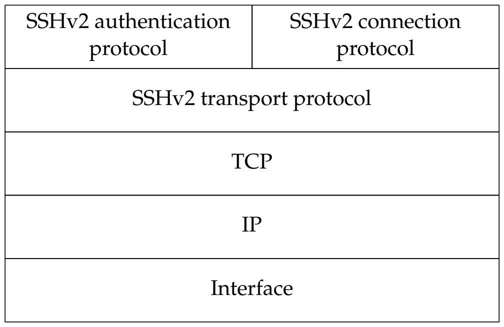

Figure 7.25 The SSHv2 Protocols Layered on the TCP/IP Stack

The SSHv2 Transport Protocol

The SSHv2 transport protocol provides a reliable, secure, full-duplex data stream between the SSH peers. *Secure* means that the data is encrypted and has strong integrity checking in place. *Full duplex* means that there is an independent data stream in each direction, so both sides can transmit data independently.

In addition to providing a secure data stream, the transport protocol is responsible for authenticating the server to the client; for negotiating the encryption, integrity, and compression methods; and for the initial and subsequent key exchanges. These methods can be negotiated independently for each side. Thus, each peer can choose its own encryption, integrity, and compression methods from the set supported by its peer.

The currently supported encryption methods are 3des-cbc; blowfish-cbc; 128, 192, and 256-bit twofish-cbc; 128, 192, and 256-bit aes-cbc; 128, 192, and 256-bit serpent-cbc; idea-cbc; 128-bit cast-cbc; and RC4. The only required method is 3des-cbc. The 128-bit aes-cbc method is recommended, and the others are optional. Notice that all the block ciphers operate in cipher block chaining mode.

The draft RFC on the SSHv2 transport protocol defines four possible MAC methods: SHA-1 HMAC, SHA-1 HMAC-96, MD5 HMAC, and MD5 HMAC-96. The SHA-1 HMAC-96 and MD5 HMAC-96 methods merely use the first 96 bits of the corresponding HMAC output. See Chapter 12 for more on SHA-1 HMAC-96 and MD5 HMAC-96. Only SHA-1 HMAC is required. SHA-1 HMAC-96 is recommended, and the MD5 methods are optional. Regardless of the method chosen, the MAC is calculated by

$$MAC = \mathrm{M}(key, sequence\ number \,\|\, m)$$

where M is the MAC method, *key* is computed from a shared secret, *m* is the unencrypted message, and *sequence number* is a 4-byte unsigned integer that has an initial value of 0 and is incremented by 1 after each packet is sent. The *sequence number* field is implicit—it is never sent. The message, *m*, includes the length, type, and payload.

Currently, the only compression method defined is ZLIB. This is the same compression used in SSHv1, and as in SSHv1, it may be indicated when SSH is used over low-bandwidth connections.

As we'll see when we examine algorithm negotiation, the encryption, integrity, and compression methods are identified by symbolic names. This makes it easy to add new algorithms without changing the protocol itself. In addition, the naming conventions make it possible for individual users or groups to define their own algorithms and name them without fear of a naming collision.

Before discussing the SSHv2 transport protocol in detail, we need to understand the binary packet protocol that it uses. The packet layout is slightly different from the SSHv1 binary packet layout in Figure 7.2. The SSHv2 binary packet layout is shown in Figure 7.26.

The *length* field is the length of the entire packet, with the exception of the length and MAC fields. Similarly, the *pad len* field is the length of the padding field that follows the payload data.

The *data* field is the payload: the SSHv2 message that the transport layer is carrying. This message can be from any of the layers. The data field is $length - pad\ len - 1$ bytes long. As with the SSHv1 binary packet protocol, each message begins with a 1-byte message type. The message types used by the transport protocol are shown in Figure 7.27. In addition, message numbers 30–49 are reserved for key-exchange packets. We will encounter these message types when we examine the key exchange protocols.

Immediately following the data is the *padding* field, which consists of pad len bytes of random data such that the length of the packet minus the MAC is a multiple of the cipher block size or 8, whichever is larger. The amount of padding must be between 4 and 255 bytes inclusive.

As discussed previously, the *MAC* field provides strong data integrity. Its length depends on the MAC method in use.

As we see from the figure, the entire packet, with the exception of the MAC, is encrypted. Note that this is different from the SSHv1 binary protocol, in which the CRC but not the length was encrypted.

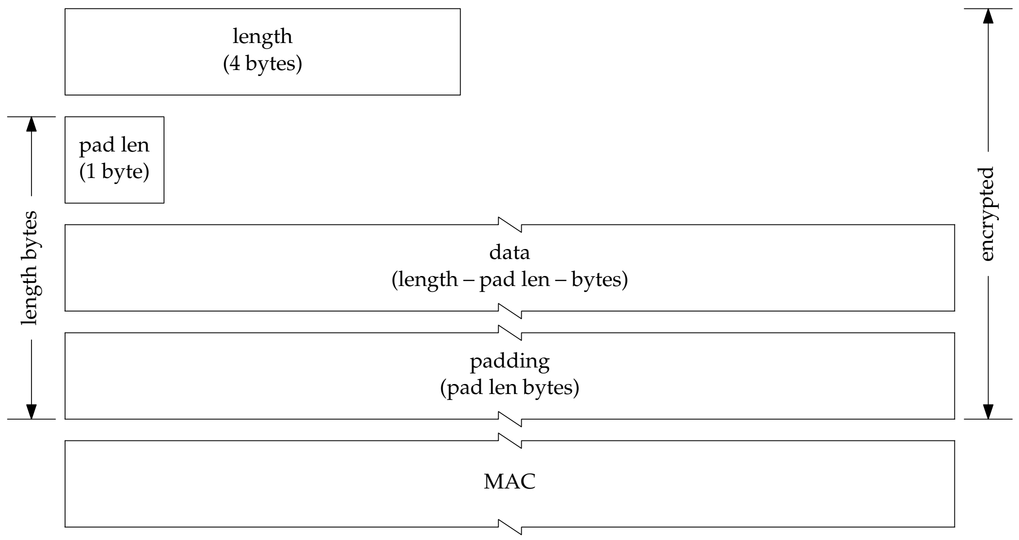

Figure 7.26 The SSHv2 Binary Packet

No.	Message Name	Message
1	SSH_MSG_DISCONNECT	terminate the connection
2	SSH_MSG_IGNORE	no op
3	SSH_MSG_UNIMPLEMENTED	message type not implemented
4	SSH_MSG_DEBUG	message contains debugging information
5	SSH_MSG_SERVICE_REQUEST	client service request
6	SSH_MSG_SERVICE_ACCEPT	server accepts service request
20	SSH_MSG_KEXINIT	algorithm negotiation
21	SSH_MSG_NEWKEYS	begin using new encryption keys

Figure 7.27 SSHv2 Transport Message Types

Algorithm Negotiation and Key Exchange

The first step in establishing the transport layer's secure data stream is for the peers to agree on the encryption, integrity, and compression algorithms to use and to agree on the keys and IVs for the encryption and integrity algorithms. Figure 7.28 shows the protocol flow for this step. The process begins with the exchange of SSH_MSG_KEXINIT messages that list the methods that each peer can support and which methods each peer prefers. After each side receives its peer's SSH_MSG_KEXINIT message, the peers will either agree on the algorithms to use or send an SSH_MSG_DISCONNECT message and terminate the connection.

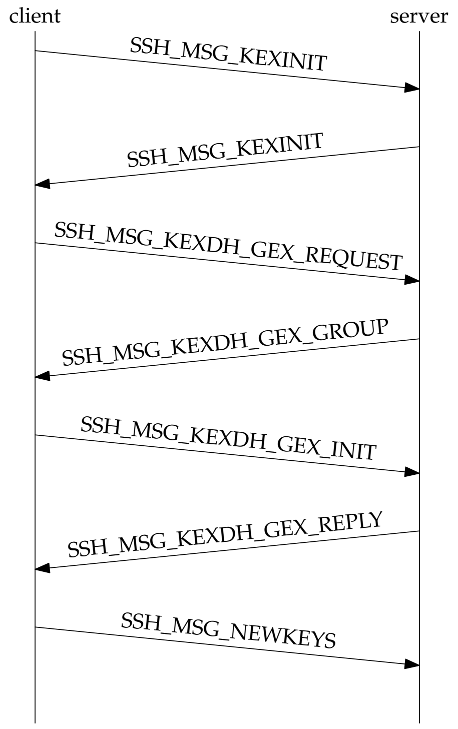

Figure 7.28 Algorithm Negotiation and Key Exchange

Currently, only two key-exchange methods are defined: Diffie-Hellman group 1 SHA-1 and Diffie-Hellman group exchange SHA-1 (DHGEX). The Diffie-Hellman group 1 SHA-1 protocol uses a fixed group with which to make its calculations, whereas the Diffie-Hellman group exchange SHA-1 protocol negotiates the group to use, based on the desired size of the prime modulus (see Chapter 3).

In Figure 7.28, the peers have agreed to use the Diffie-Hellman group exchange SHA-1 protocol. They begin by deciding on the group to use and then perform the Diffie-Hellman exchange. We will study these messages in detail shortly.

Finally, the client sends an SSH_MSG_NEWKEYS message to the server. This message signals that the peers should begin using the previously agreed on encryption, integrity, and compression algorithms. As we'll discuss later, either peer may initiate a new key exchange at any time. After the negotiation, one peer will send the SSH_MSG_NEWKEYS message to start the use of the new keys. The format of the SSH_MSG_KEXINIT message is shown in Figure 7.29.

The *type* field contains the message number: SSH_MSG_KEXINIT, in this case. The *cookie* field is not really a cookie, in the sense that the receiver will return it to the sender, but is simply a sequence of 16 random bytes. Because this field will be used in the

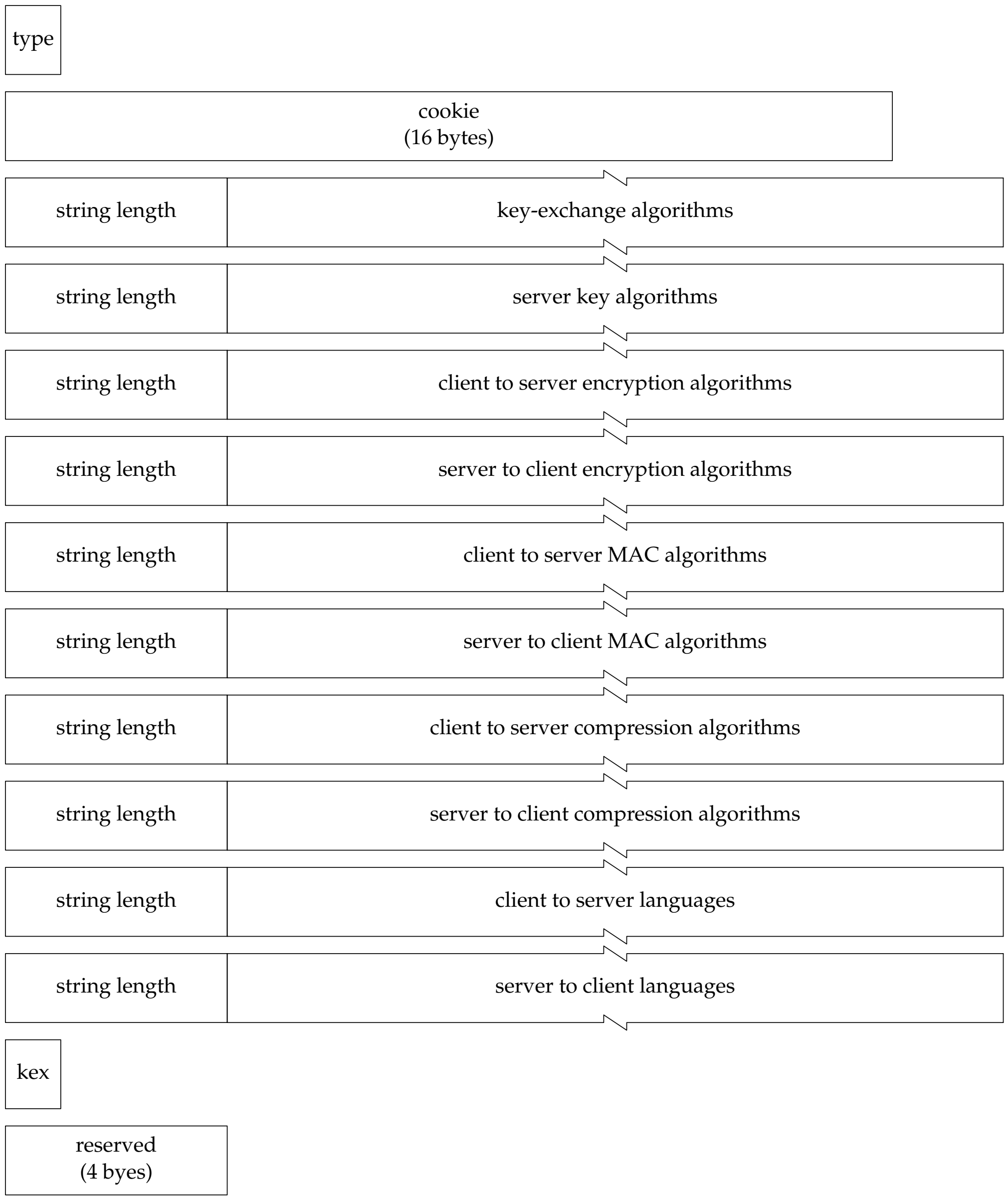
type
cookie
(16 bytes)
string length
key-exchange algorithms
string length
server key algorithms
string length
client to server encryption algorithms
string length
server to client encryption algorithms
string length
client to server MAC algorithms
string length
server to client MAC algorithms
string length
client to server compression algorithms
string length
server to client compression algorithms
string length
client to server languages
string length
server to client languages
kex
reserved
(4 byes)

Figure 7.29 The SSH_MSG_KEXINIT Message

key-generation process and because each side sends an independent value for this field, the cookie serves to ensure that neither side can completely determine the keys that the connection will use.

The next eight fields are lists of comma-separated algorithms that the sender supports. The lists are sorted in order of preference. We will see an example of these lists shortly. Most of these fields are self-explanatory. The *host key algorithm* field lists the types of host keys used for server authentication that the sender supports. Recall that in SSHv1, this was always an RSA key. SSHv2 supports additional methods.

The next two fields are lists of comma-separated language tags. The use of these tags is explained in RFC 3066 [Alvestrand 2001]. These lists are empty unless they are known to be needed by the sending party.

The *kex* field is a Boolean field that indicates whether a guessed key-exchange packet follows. The SSHv2 transport protocol allows each peer to guess what key-exchange protocol its peer will request and to send the initial key-exchange message immediately. If a guessed initial key-exchange packet follows, this field will be TRUE.

This capability adds complexity for little apparent benefit—never a good tradeoff in security software. OpenSSH always sets the kex field to FALSE but does, of course, honor its use by a peer.

Because no encryption is used until after the first SSH_MSG_NEWKEYS message, we can capture the entire algorithm and key-exchange protocol exchange of Figure 7.28. Here is a capture of an SSH_MSG_KEXINIT message:

```
1     15:12:17.920079 localhost.32775 > localhost.ssh: P 23:567(544)
      ack 24 win 32767 <nop,nop,timestamp 2036746 2036746> (DF)
1.1       4500 0254 001b 4000 4006 3a87 7f00 0001    E..T..@.@.:.....
1.2       7f00 0001 8007 0016 fded 9b30 fdd3 3e71    ...........0..>q
1.3       8018 7fff 2456 0000 0101 080a 001f 140a    ....$V..........
1.4       001f 140a 0000 021c 0914 752b cb0f 1554    ..........u+...T
1.5       933c a4d0 c8d9 f222 cbbb 0000 003d 6469    .<.....".....=di
1.6       6666 6965 2d68 656c 6c6d 616e 2d67 726f    ffie-hellman-gro
1.7       7570 2d65 7863 6861 6e67 652d 7368 6131    up-exchange-sha1
1.8       2c64 6966 6669 652d 6865 6c6c 6d61 6e2d    ,diffie-hellman-
1.9       6772 6f75 7031 2d73 6861 3100 0000 0f73    group1-sha1....s
1.10      7368 2d72 7361 2c73 7368 2d64 7373 0000    sh-rsa,ssh-dss..
1.11      0066 6165 7331 3238 2d63 6263 2c33 6465    .faes128-cbc,3de
1.12      732d 6362 632c 626c 6f77 6669 7368 2d63    s-cbc,blowfish-c
1.13      6263 2c63 6173 7431 3238 2d63 6263 2c61    bc,cast128-cbc,a
1.14      7263 666f 7572 2c61 6573 3139 322d 6362    rcfour,aes192-cb
1.15      632c 6165 7332 3536 2d63 6263 2c72 696a    c,aes256-cbc,rij
1.16      6e64 6165 6c2d 6362 6340 6c79 7361 746f    ndael-cbc@lysato
1.17      722e 6c69 752e 7365 0000 0066 6165 7331    r.liu.se...faes1
1.18      3238 2d63 6263 2c33 6465 732d 6362 632c    28-cbc,3des-cbc,
1.19      626c 6f77 6669 7368 2d63 6263 2c63 6173    blowfish-cbc,cas
1.20      7431 3238 2d63 6263 2c61 7263 666f 7572    t128-cbc,arcfour
1.21      2c61 6573 3139 322d 6362 632c 6165 7332    ,aes192-cbc,aes2
1.22      3536 2d63 6263 2c72 696a 6e64 6165 6c2d    56-cbc,rijndael-
1.23      6362 6340 6c79 7361 746f 722e 6c69 752e    cbc@lysator.liu.
1.24      7365 0000 0055 686d 6163 2d6d 6435 2c68    se...Uhmac-md5,h
1.25      6d61 632d 7368 6131 2c68 6d61 632d 7269    mac-sha1,hmac-ri
1.26      7065 6d64 3136 302c 686d 6163 2d72 6970    pemd160,hmac-rip
1.27      656d 6431 3630 406f 7065 6e73 7368 2e63    emd160@openssh.c
1.28      6f6d 2c68 6d61 632d 7368 6131 2d39 362c    om,hmac-sha1-96,
1.29    686d 6163 2d6d 6435 2d39 3600 0000 5568    hmac-md5-96...Uh
1.30    6d61 632d 6d64 352c 686d 6163 2d73 6861    mac-md5,hmac-sha
1.31    312c 686d 6163 2d72 6970 656d 6431 3630    1,hmac-ripemd160
1.32    2c68 6d61 632d 7269 7065 6d64 3136 3040    ,hmac-ripemd160@
1.33    6f70 656e 7373 682e 636f 6d2c 686d 6163    openssh.com,hmac
1.34    2d73 6861 312d 3936 2c68 6d61 632d 6d64    -sha1-96,hmac-md
1.35    352d 3936 0000 0009 6e6f 6e65 2c7a 6c69    5-96....none,zli
1.36    6200 0000 096e 6f6e 652c 7a6c 6962 0000    b....none,zlib..
1.37    0000 0000 0000 0000 0000 0000 0000 0000    ................
1.38    0000 0000                                  ....
```

On line 1.4 the length, pad len, and type fields are set in boldface. The length field is 540 (0x21c) bytes, which agrees with the total TCP segment length of 544 on line 1. Next is the pad len field, set to 9. The padding is the last 9 bytes of zeros in the packet; there is no MAC field yet—that field will be present after the SSH_MSG_NEWKEYS message. Finally, the message type is 20 (0x14), as expected.

The 16 bytes following the message type are the cookie. The lists of algorithms follow the cookie. For example, the list of key-exchange algorithms is on lines 1.5–1.9. We see that OpenSSH supports the diffie-hellman-group-exchange-sha1 and diffie-hellman-group1-sha1 methods, as we discussed previously.

On lines 1.26 and 1.27 is an example of a "private algorithm." The MAC method `hmac-ripemd160@openssh.com` is defined by the OpenSSH organization. The `@openssh.com` identifies the method name space as belonging to the `openssh.com` domain. Other examples of private algorithms are found in the lists of encryption algorithms—see lines 1.15–1.17, for instance. Peers that don't know about these private methods will, of course, ignore them in the method-selection process. Methods assigned by IETF do not have the `@` and domain name appended.

SSHv2 Diffie-Hellman Key Exchange Protocol

Returning to Figure 7.28, we see that the peers have agreed to use the Diffie-Hellman group exchange protocol for key exchange. The process begins with the client sending the server an SSH_MSG_KEXDH_GEX_REQUEST message, which specifies the maximum, minimum, and preferred prime size for the Diffie-Hellman group. The format of this message is shown in Figure 7.30. The *type* field is set to 34, indicating an SSH_MSG_KEXDH_REQUEST message. The *min*, *preferred*, and *max* fields are 32-bit unsigned integers specifying the minimum, preferred, and maximum size in bits of the prime for the group.

The server selects a group of the appropriate size and informs the client of its choice with an SSH_MSG_KEXDH_GEX_GROUP message. This message (type 31) contains two multiprecision integers containing the chosen prime, p, and the corresponding generator g. After the client receives this message, both peers know the Diffie-Hellman group to use.

The next two messages complete the Diffie-Hellman key exchange. The client chooses a random number r_C such that $1 < r_C < (p-1)/2$, calculates $e = g^{r_C} \bmod p$, and sends it to the server as a multiprecision integer in an SSH_MSG_KEXDH_GEX_INIT (type 32) message.

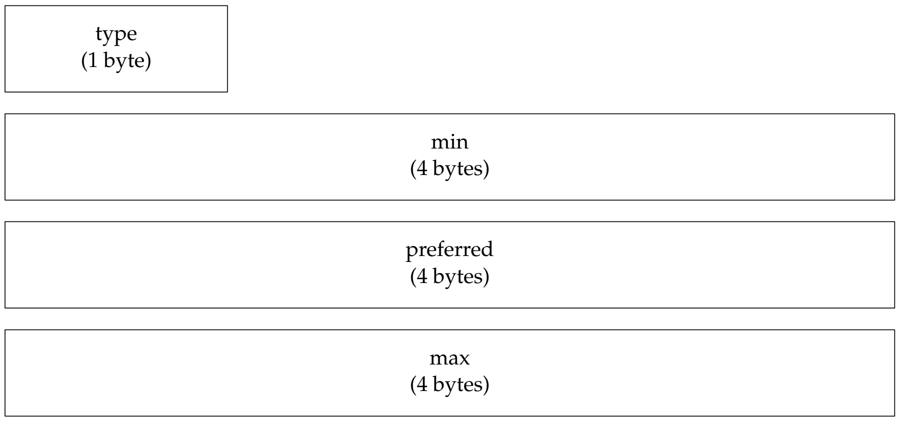

Figure 7.30 The SSH_MSG_KEXDH_REQUEST Message

On receipt of the SSH_MSG_KEXDH_GEX_INIT message, the server chooses its own random number r_S, calculates $f = g^{r_S} \bmod p$, and sends this value along with its server key and the signature of the exchange hash, H, to the client in an SSH_MSG_KEXDH_GEX_REPLY message (Figure 7.31). We discuss the exchange hash shortly.

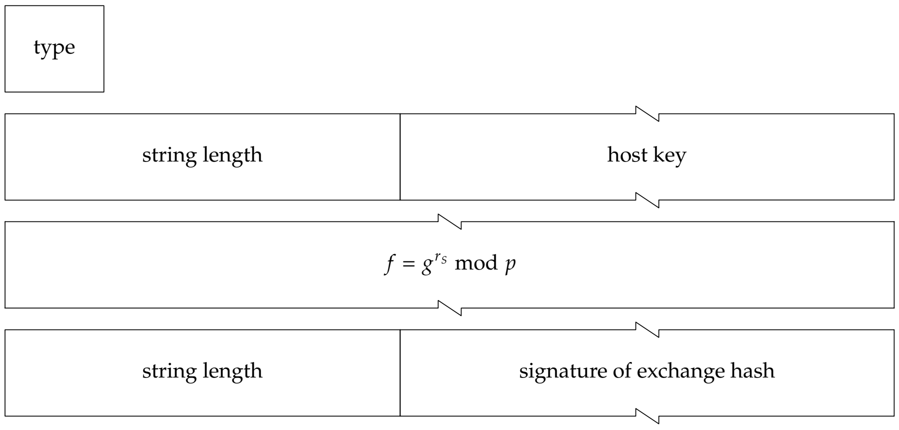

Figure 7.31 The SSH_MSG_KEXDH_GEX_REPLY Message

The SSH_MSG_KEXDH_GEX_REPLY message serves two purposes. First, it provides the client with its part of the Diffie-Hellman calculation that both peers will use to derive a shared secret. This is the value f. As we saw in Chapter 3, the client calculates $K = f^{r_C}$, and the server calculates the same value independently as $K = e^{r_S}$.

Second, the message provides server authentication. By including the host key, the message allows the client to identify the server from its database of server keys. The signature of the exchange hash, H, allows the client to verify that the server possesses the private key corresponding to the public key in the message.

The Exchange Hash

We've already seen that the exchange hash plays a role in server authentication. The exchange hash also plays an important role in key generation and the prevention of replay attacks. Finally, the initial exchange hash—the one from the first key exchange—serves as the session ID.

Recall that the key-exchange algorithms include a hash method. In both of the currently defined key-exchange methods, this is the SHA-1 hash. In what follows, we'll call this hash function $\mathrm{h}(m)$.

The exchange hash is calculated as

$$H = \mathrm{h}(S)$$

where S is the concatenation of the quantities shown in Figure 7.32.

Type	Value	Explanation	DHGEX Only
string	V_C	client's version string (CR and NL excluded)	
string	V_S	server's version string (CR and NL excluded)	
string	I_C	payload of client's SSH_MSG_KEXINIT message	
string	I_S	payload of server's SSH_MSG_KEXINIT message	
string	K_S	host key	
integer	*min*	minimum group size	●
integer	n	preferred group size	●
integer	*max*	maximum group size	●
multiprec. int.	p	prime	●
multiprec. int.	e	client's Diffie-Hellman calculation	
multiprec. int.	f	server's Diffie-Hellman calculation	
multiprec. int.	K	shared secret	

Figure 7.32 Quantities in the Exchange Hash

Note that the quantities that make up the exchange hash are available to both the client and server. This fact allows the client to verify the server's signature of the exchange hash and both sides to independently generate the session keys.

Before leaving our discussion of key exchange, let's turn to the other method: Diffie-Hellman group 1 with SHA-1. The method is similar to Diffie-Hellman group exchange but uses a fixed group, so there is no group-size negotiation.

Using a single fixed group, as this method does, leaves the Diffie-Hellman exchange open to a precomputation attack. Most of the work involved in solving the discrete logarithm problem involves precomputation on the group and its generator defined by the values of g and p [Herzog 1999]. Using a fixed group makes it possible to perform these computations once offline and then use them repeatedly in different attacks.

It uses just two messages: the SSH_MSG_KEXDH_INIT (type 30) message, which is identical to the SSH_MSG_KEXDH_GEX_INIT message from the group exchange method, and the SSH_MSG_KEXDH_REPLY (type 31) message, which is identical to the SSH_MSG_KEXDH_GEX_REPLY message.

The group exchange method reuses message numbers from the group 1 method. Although this seems unnecessarily confusing, it allows new methods to be added without having to reserve message number ranges in advance. The first message in the exchange identifies the method, and the subsequent messages in the method can use any number in the range reserved for key exchange. We will see this reuse again in the authentication protocol.

Key Generation

The types of session keys and their calculation are shown in Figure 7.33. The peers will generate whichever keys are required for the chosen session algorithms. For example, the initial IVs are not required if the peers are using a stream cipher, such as RC4.

Key	Calculation
client to server initial IV	h(K || H || "A" || SID)
server to client initial IV	h(K || H || "B" || SID)
client to server encryption	h(K || H || "C" || SID)
server to client encryption	h(K || H || "D" || SID)
client to server integrity	h(K || H || "E" || SID)
server to client integrity	h(K || H || "F" || SID)

K = shared secret
H = exchange hash
SID = session ID

Figure 7.33 Session Key Generation

With the SHA-1 hash, each of the calculations shown in Figure 7.33 will generate 160 bits of key material. If additional bits are required, they are generated by hashing the concatenation of K, H, and the key so far. This process is continued until a sufficient number of bits are generated. For example, to extend the bits for the client to server encryption key, we would calculate

$$key_1 = \mathrm{h}(K \,\|\, H \,\|\, \text{"C"} \,\|\, SID)$$

$$key_2 = \mathrm{h}(K \,\|\, H \,\|\, key_1)$$

$$key_3 = \mathrm{h}(K \,\|\, H \,\|\, key_1 \,\|\, key_2)$$

$$\ldots$$

$$key = key_1 \,\|\, key_2 \,\|\, key_3 \,\|\, \cdots$$

Either peer can initiate a rekeying by sending an SSH_MSG_KEXINIT message. This process is the same as the initial algorithm negotiation and key generation, so the peers can even renegotiate the algorithms they will use, if desired. Because the peers will use new cookies for the SSH_MSG_KEXINIT messages and will calculate a new shared secret, the newly generated keys will be distinct from the old keys.

Rekeying after transmitting a certain amount of data is always a good idea because it makes it more difficult for an attacker to gather enough ciphertext for an effective cryptanalysis. With SSHv2, there is a more immediate reason for rekeying. Recall that each packet's MAC contains an implicit sequence number that is based on a 32-bit counter. If this counter is allowed to wrap while using the same keys, the session becomes subject to replay, information leakage, and out-of-order delivery attacks [Bellare, Kohno, and Namprempre 2002]. The draft SSHv2 transport protocol specification recommends that a session be rekeyed after each gigabyte of data or after an hour of connection time without rekeying.

During the rekeying operation, the peers use the old keys and algorithms. When the rekeying operation is complete, an SSH_MSG_NEWKEYS message signals the peers to begin using the new keying material and algorithms. The SSH specification is unclear on how rekeying should take place, and many implementations don't support it [Gutmann 2005].

Service Requests

After the key exchange, the transport protocol has established its secure data stream and is now ready to start carrying data for the authorization and connection protocols. To request one of these protocols, the client sends the server an SSH_MSG_SERVICE_REQUEST message containing the service name as a string. Currently, the specification defines the two services ssh-userauth and ssh-connection.

If the server supports the service request and is willing to let the client invoke it, the server will send the client an SSH_MSG_SERVICE_ACCEPT message, which echos the service name as a string. If the server does not support the service or is unwilling to let the client invoke it, it will send the client an SSH_MSG_DISCONNECT message.

The SSHv2 Authentication Protocol

When the authentication protocol begins running, the transport protocol has already authenticated the server to the client. The authentication protocol's purpose is to authenticate the user—and sometimes the client host—to the server.

Four user authentication methods are defined: public key, password, keyboard interactive, and host based. The host-based method is similar to the rhosts/RSA method from SSHv1 and is inappropriate in most situations requiring serious security. We will not discuss this method further.

The public key and password methods are functionally similar to their counterparts in SSHv1, although the protocols implementing them are different. Keyboard interactive is a general method that encompasses any authentication algorithm that can be implemented on the client side by user keyboard interaction. The advantage of this method is that new algorithms can be implemented on the server side without requiring changes to the clients. This method can be used to implement challenge/response and one-time-password schemes in SSHv2 but finds its main use in the pluggable authentication modules (PAM) framework.

The general authentication message types are shown in Figure 7.34. We'll discuss other, method-specific messages, when we cover the method in question.

No.	Message Name	Message
50	SSH_MSG_USERAUTH_REQUEST	authorization initiation
51	SSH_MSG_USERAUTH_FAILURE	server rejects authentication attempt
52	SSH_MSG_USERAUTH_SUCCESS	server accepts authentication
53	SSH_MSG_USERAUTH_BANNER	login banner from the server

Figure 7.34 General Authentication Messages

All authentication methods are initiated by the client sending the server an SSH_MSG_USERAUTH_REQUEST message. The general format of this message is shown in Figure 7.35. The method-specific data varies with the particular authentication method.

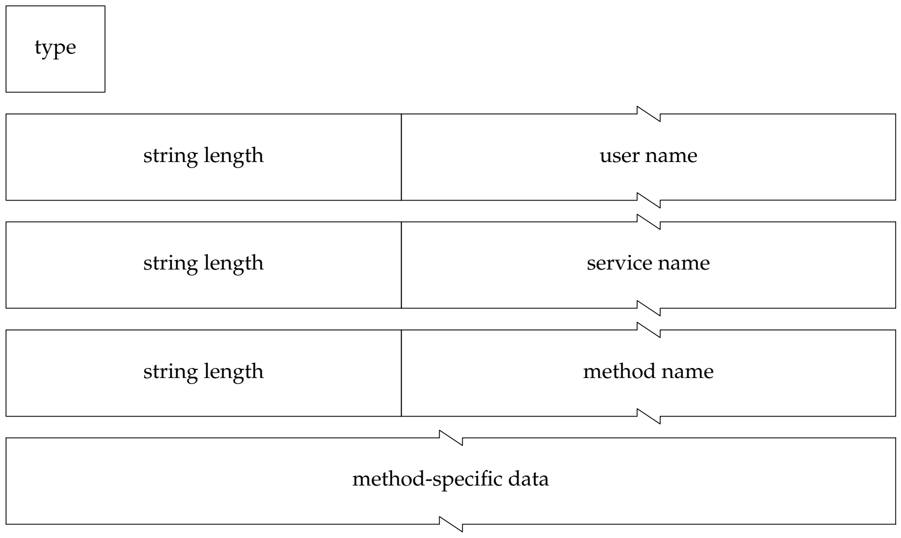

Figure 7.35 General Format of the User Authentication Request

The *type* field is set to SSH_MSG_USERAUTH_REQUEST. The particular type of request is given as an ASCII string in the *method name* field.

The *user name* field contains the user's name encoded as a UTF-8 string. The *service name* contains the name of the service for which the user is asking to be authenticated. This field is encoded as an ASCII string.

The SSH_MSG_USERAUTH_FAILURE message, shown in Figure 7.36, is sent by the server to indicate partial success or complete failure of the authentication.

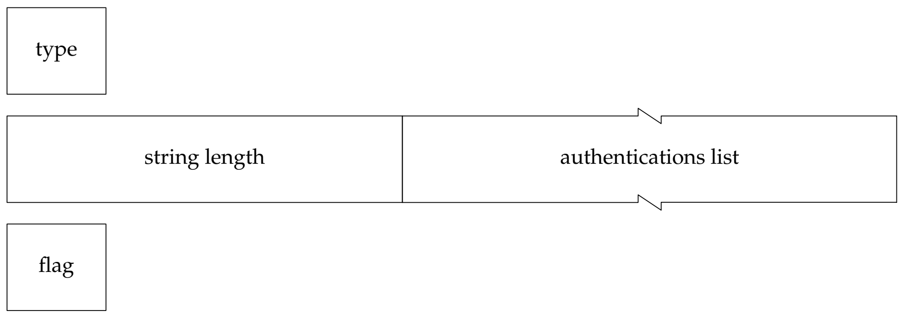

Figure 7.36 The SSH_MSG_USERAUTH_FAILURE Message

The *authentications list* field is a list of authentication methods that the client can try instead of the failed method. The *flag* field provides additional message-specific semantics.

When the client successfully completes authentication to the server's satisfaction, the server responds with an SSH_MSG_USERAUTH_SUCCESS message, which ends the authentication protocol successfully. This message contains no arguments.

The final general authentication message is the SSH_MSG_USERAUTH_BANNER message. The server sends this message to request that the client display a login banner to the user. The client may decide whether to display the message. The SSH_MSG_USERAUTH_BANNER contains two strings: a UTF-8 encode string with the message itself and an RFC 3066 language tag.

The "none" Authentication Method

In the SSHv1 protocol, the server sends the client a list of supported authentication methods in the SSH_SMSG_PUBLIC_KEY message at the beginning of the protocol exchange. In SSHv2, the client requests a list of supported authentication methods by sending an SSH_MSG_USERAUTH_REQUEST message specifying the "none" authentication message. If the server does not require authentication, it can reply with an SSH_MSG_USERAUTH_SUCCESS message, but the normal procedure is to return an SSH_MSG_USERAUTH_FAILURE message with a list of supported authentication methods in the authentications list field. There is no message-specific data in the SSH_MSG_USERAUTH_REQUEST message when the method name is set to none.

The Public Key Authentication Method

The public key authentication method is similar to the RSA method from SSHv1 but is generalized to support any signature method. As we see from the `tcpdump` of the SSH_MSG_KEXINIT, OpenSSH supports RSA and DSS signatures.

The client requests a public key authentication by sending an SSH_MSG_USERAUTH_REQUEST message with the method name set to "publickey." The method-specific data, shown in Figure 7.37, includes three strings containing the name of the signature algorithm, the public key, and the signature itself. The *flag* field is set to TRUE.

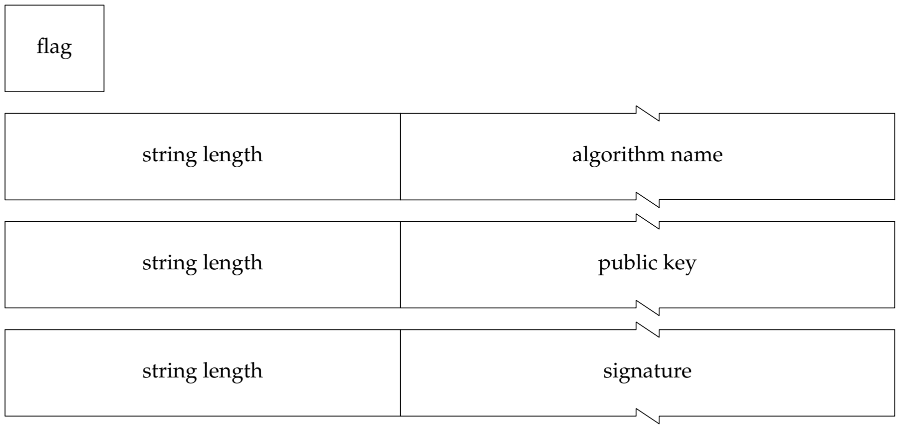

Figure 7.37 Method-Specific Data for the Public Key Method

The signature is taken over the data shown in Figure 7.38. Note that the public key authentication method is different from the challenge/response protocol used in SSHv1's RSA authentication. By sending the signature along with the request, the protocol saves two packet exchanges.

Type	Data
string	session ID
byte	SSH_MSG_USERAUTH_REQUEST
string	user name
string	service
string	"publickey"
Boolean	TRUE
string	algorithm name
string	public key

Figure 7.38 Data for the Public Key Signature

If the public key is not valid for this user or if the signature does not validate, the server will respond with an SSH_MSG_USERAUTH_FAILURE message. If both tests succeed and no further authentication is required, the server will respond with an SSH_MSG_USERAUTH_SUCCESS message, and the authentication will be complete.

Computing the signature is computationally expensive and may require user interaction if the private key is protected with a password. The protocol allows the client to verify that the public key and algorithm are acceptable to the server before asking the user for the password or calculating the signature. To do this, the client sends an SSH_MSG_USERAUTH_REQUEST message with the method-specific data, as shown in

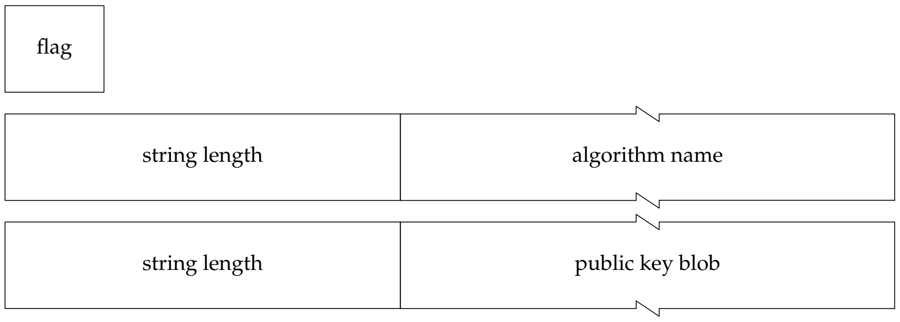

Figure 7.39 Message-Specific Data for the Public Key Test Message

Figure 7.39. The *flag* field is set to FALSE to distinguish this message from an actual public key authentication request. The *public key blob* field contains the public key and, optionally, certificates or other information.

The Password Authentication Method

From the user's perspective, the SSHv2 password authentication is identical to the SSHv1 version, except that the client or server may request the user to enter a new password. This last facility is usually used by the server when a user's password has expired. The client requests a password authentication by sending the server an SSH_MSG_USERAUTH_REQUEST message with the method name set to "password." The message-specific data consists of a flag set to FALSE and a string containing the password encoded in UTF-8.

Normally, the server will respond with an SSH_MSG_USERAUTH_SUCCESS or SSH_MSG_USERAUTH_FAILURE message but can also request a new password by sending an SSH_MSG_USERAUTH_PASSWD_CHANGEREQ message (Figure 7.40). The *type* field is set to SSH_MSG_USERAUTH_PASSWD_CHANGEREQ (60). The *prompt* field contains the user prompt in the language defined by the *language tag* field.

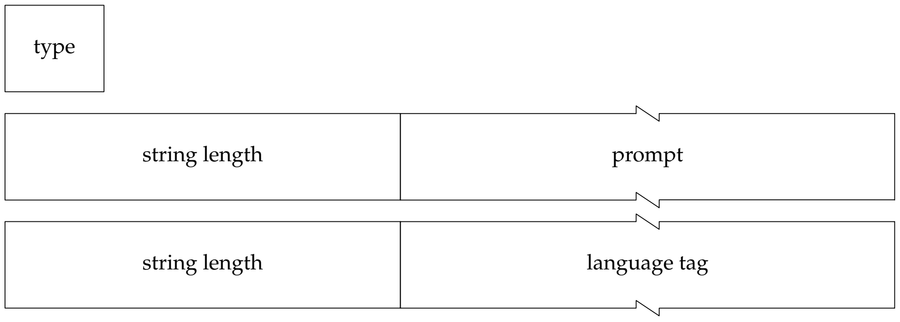

Figure 7.40 The Change Password Request Message

At this point, the client can either try another authentication message or retry the password authentication, using the message-specific data shown in Figure 7.41. In this case, the *flag* field is set to TRUE.

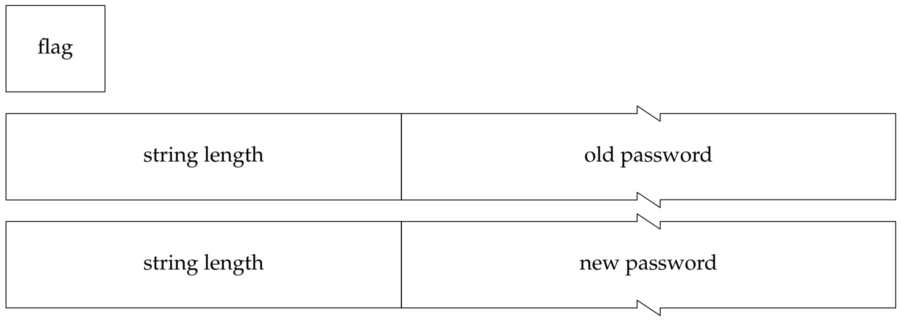

Figure 7.41 Method-Specific Data for the Change Password Message

The server responds to the client's change password message with one of the following: SSH_MSG_USERAUTH_SUCCESS, indicating that the password was changed and that the authentication was successful; SSH_MSG_USERAUTH_FAILURE, indicating that the password was changed but that more authentication is needed (partial success TRUE) or that the password was not changed (partial success FALSE), due to a bad old password or password changing not being supported; or SSH_MSG_USERAUTH_CHANGEREQ, indicating that the new password was not acceptable.

The Keyboard Interactive Authentication Method

The final authentication method that we look at is the keyboard interactive method, a generalized method that can support any authentication scheme that requires only user keyboard input. One could, in principle, use this method to do ordinary password authentication, but its main use is in challenge/response and one-time-password schemes.

The client requests keyboard interactive authentication by sending the server an SSH_MSG_USERAUTH_REQUEST message with the method type set to "keyboard-interactive," and the message-specific data shown in Figure 7.42. The *language tag* field, if not empty, indicates the language encoding that the user prompts should be in. The *submethods* field is a list of authentication methods that the client would like to try. The server treats this field as a hint and is free to try whatever method it chooses.

The server may respond with an SSH_MSG_USERAUTH_FAILURE message if it does not support keyboard interactive authentication or an SSH_MSG_USERAUTH_SUCCESS message if no further authentication is required. Normally, however, the server will respond with the SSH_MSG_USERAUTH_INFO_REQUEST (60) message shown in Figure 7.43.

The *name* and *instruction* fields are displayed to the user. Either or both fields may be empty. The *language tag* field is intended to indicate to the client which language

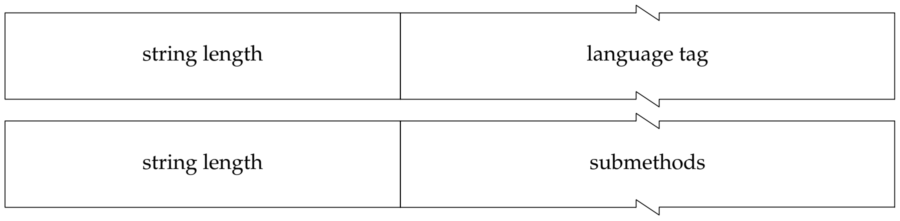

Figure 7.42 Message-Specific Data for the Keyboard Interactive Message

encoding should be used for the messages it displays to the users. The draft specification recommends that this field be empty and that the peers use the encodings negotiated during key exchange.

The *number of prompts* field tells the client how many prompt/echo pairs follow. These are labeled $prompt_1 \cdots prompt_n$ and $echo_1 \cdots echo_n$ in Figure 7.43. Each $prompt_i$ field contains a prompt for the user ("`Password: `", for example), and the corresponding $echo_i$ flag tells the client whether to echo the response to the user's terminal.

On receipt of the SSH_MSG_USERAUTH_INFO_REQUEST message, the client displays the name and instruction fields on the user's display and then prompts the user with each of the included prompts, either echoing the response or not, as indicated by the corresponding echo flag. The server can send additional SSH_MSG_USERAUTH_INFO_REQUEST messages but only after the client responds to the current one.

After prompting the user and gathering the replies, the client will respond with an SSH_MSG_USERAUTH_INFO_RESPONSE message (Figure 7.44). The *number of responses* field must contain the same number as the server's number of prompts field. The $response_i$ fields carry the user's response to the server's prompts.

The Connection Protocol

After authentication, SSH starts the ssh-connection service, which handles remote shells, remote execution of commands, X11 forwarding, and TCP/IP port forwarding. With the exception of file transfer utilities, such as `scp`, these services are implemented similarly to the way they were in SSHv1.

As in SSHv1, multiple channels can be multiplexed onto the secure data stream provided by the transport protocol. As a result, there are two classes of messages in the connection protocol: channel messages, which apply to a particular channel, and global messages, which are not associated with a channel. The message types and numbers are given in Figure 7.45.

The global request message has the general form shown in Figure 7.46. The *type* field is set to SSH_MSG_GLOBAL_REQUEST, and the specific request is carried as an ASCII string in the *request name* field. The *want reply* field is a Boolean that's set to TRUE if the sender wants a reply to the request and FALSE otherwise. When the want-reply flag is TRUE, the peer will respond with either an SSH_MSG_REQUEST_SUCCESS or an SSH_MSG_REQUEST_FAILURE message. Both of these messages normally

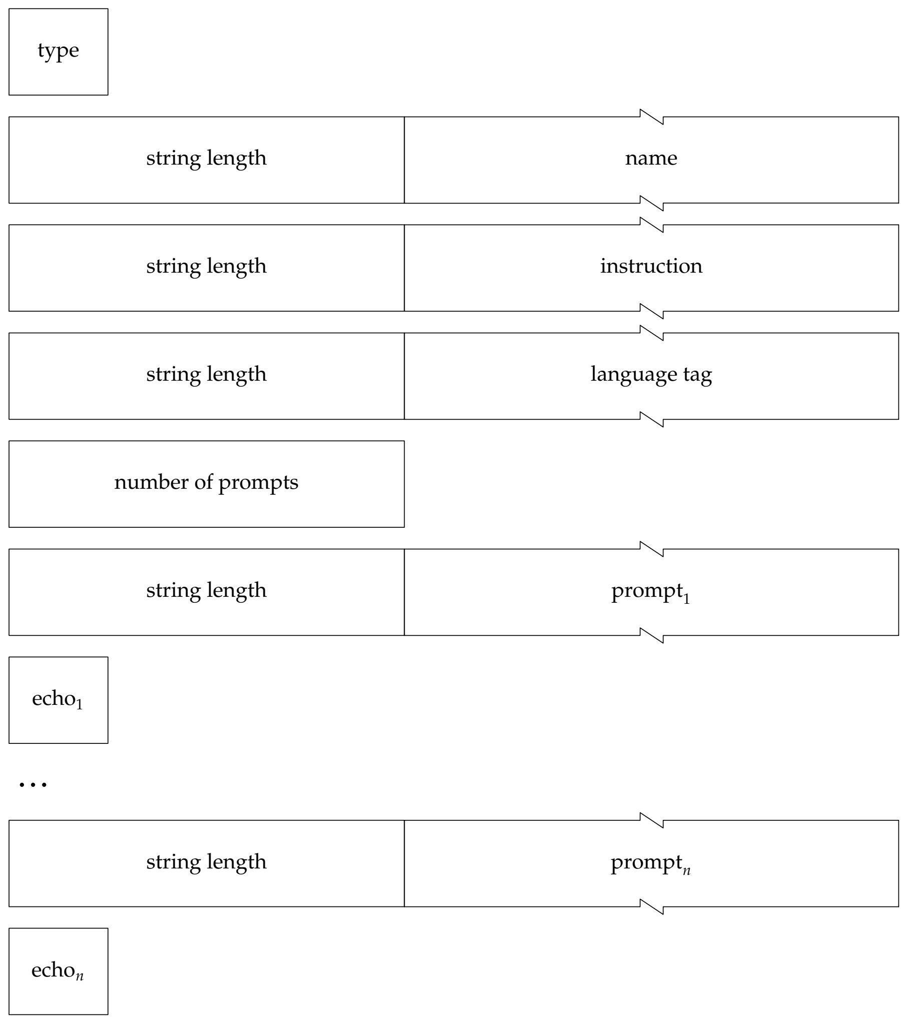

Figure 7.43 The SSH_MSG_USERAUTH_INFO_REQUEST Message

have only a type field, but the SSH_MSG_REQUEST_SUCCESS message may include data specific to the request that it is acknowledging.

The SSH_MSG_CHANNEL_OPEN, SSH_MSG_CHANNEL_OPEN_CONFIRMATION, and SSH_MSG_CHANNEL_OPEN_FAILURE messages have similar general forms. The SSH_MSG_CHANNEL_OPEN message, shown in Figure 7.47, requests that the peer allocate a new data channel. As we've seen before, each peer will have its own channel number. The *sender channel* field is the number that the message sender has assigned to the channel. The *channel type* field is the type of channel that the sender is requesting. This field is encoded in ASCII.

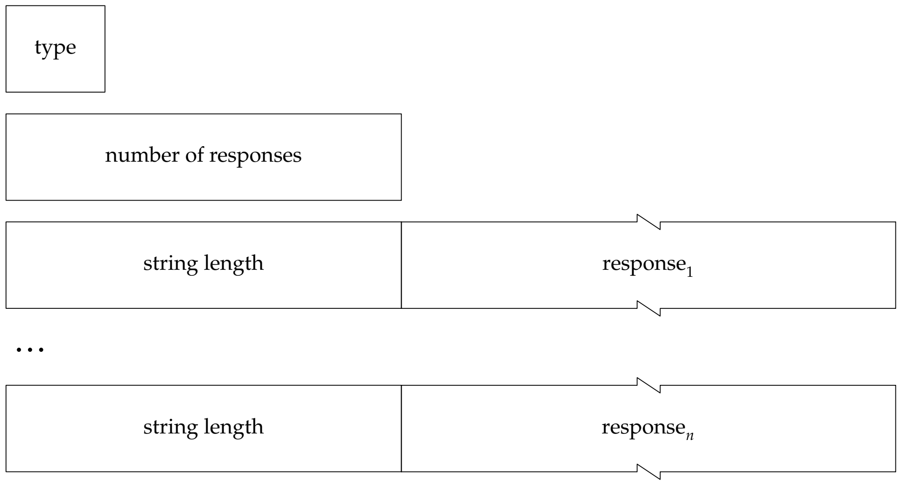

Figure 7.44 The SSH_MSG_USERAUTH_INFO_RESPONSE Message

No.	Message Name	Message
80	SSH_MSG_GLOBAL_REQUEST	global message
81	SSH_MSG_REQUEST_SUCCESS	global message succeeded
82	SSH_MSG_REQUEST_FAILURE	global message failed
90	SSH_MSG_CHANNEL_OPEN	channel open request
91	SSH_MSG_CHANNEL_OPEN_CONFIRMATION	channel open request succeeded
92	SSH_MSG_CHANNEL_OPEN_FAILURE	channel open failed
93	SSH_MSG_CHANNEL_WINDOW_ADJUST	open flow control window
94	SSH_MSG_CHANNEL_DATA	data-bearing message for a channel
95	SSH_MSG_CHANNEL_EXTENDED_DATA	extended channel data (STDERR, e.g.)
96	SSH_MSG_CHANNEL_EOF	end of file for channel data
97	SSH_MSG_CHANNEL_CLOSE	channel close request
98	SSH_MSG_CHANNEL_REQUEST	channel-specific request
99	SSH_MSG_CHANNEL_SUCCESS	channel request succeeded
100	SSH_MSG_CHANNEL_FAILURE	channel request failed

Figure 7.45 SSHv2 Channel Messages

The *initial window size* field is used for flow control. The *maximum packet size* field indicates the largest packet that the sender is willing to receive from its peer.

If the peer agrees to open the channel, it will respond with an SSH_MSG_CHANNEL_OPEN_CONFIRMATION message (Figure 7.48). The *recipient channel* is the channel number that the original sender assigned in the SSH_MSG_CHANNEL_OPEN message. The *sender channel* is the channel number that *this* sender is assigning. The other fields are identical to those in the SSH_MSG_CHANNEL_OPEN message.

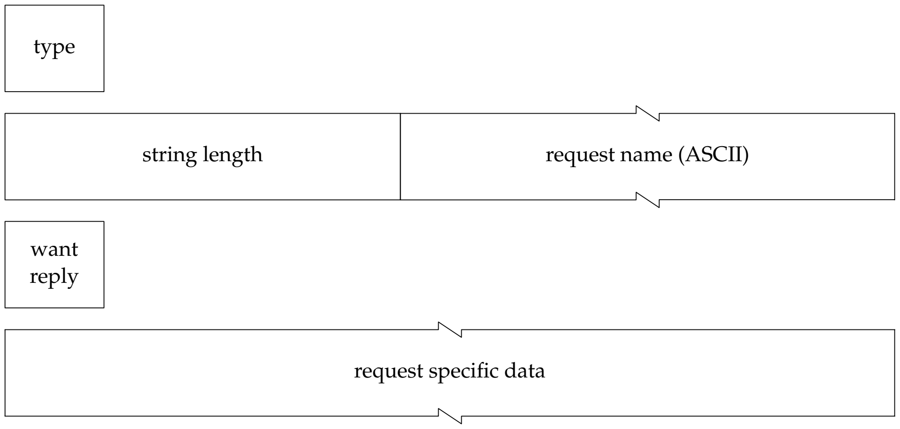

Figure 7.46 General Form of the SSH_MSG_GLOBAL_REQUEST Message

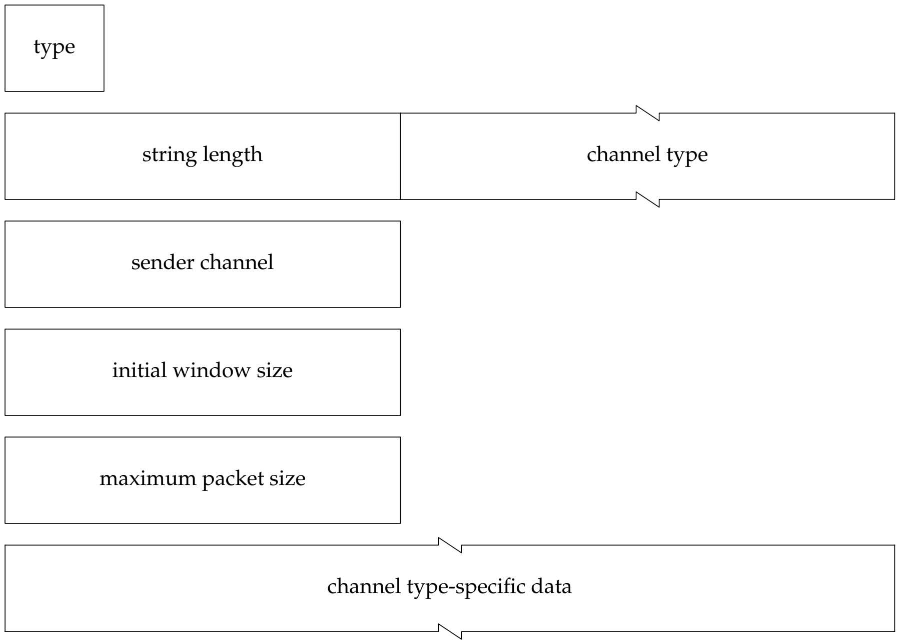

Figure 7.47 General Form of the SSH_MSG_CHANNEL_OPEN Message

If the peer cannot or is unwilling to open the channel, it will reply with the SSH_MSG_CHANNEL_OPEN_FAILURE message, shown in Figure 7.49. The *reason code* field is a numeric value giving the reason for the failure. Currently defined values

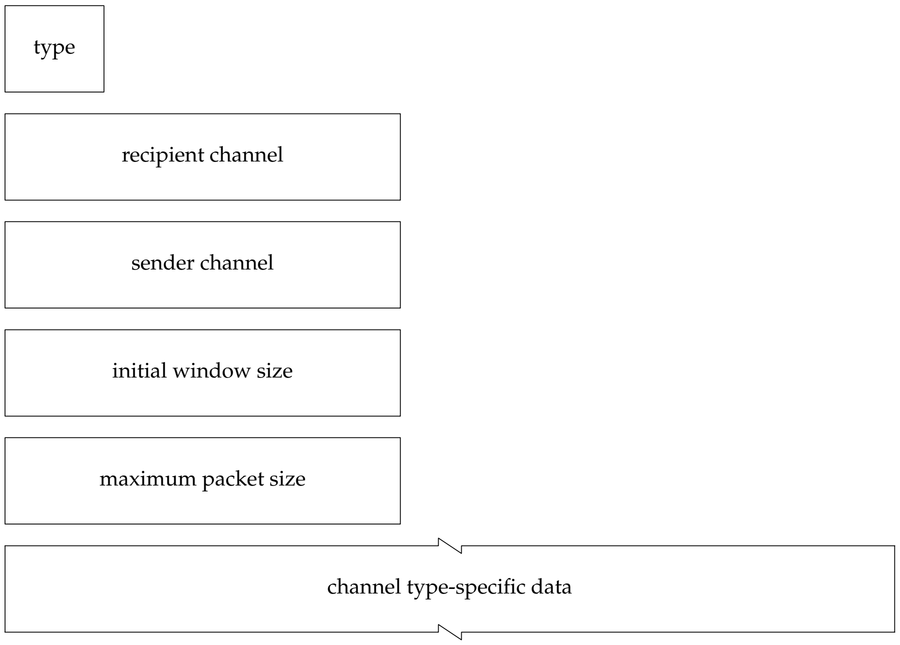

Figure 7.48 General Form of the SSH_MSG_CHANNEL_OPEN_CONFIRMATION Message

are given in Figure 7.50. The *textual information* field gives further information about the failure. It is intended for human consumption. The *language tag* field indicates the language for the textual information field.

Data Transfer

Data transfer in SSHv2 is flow controlled in a manner vaguely reminiscent of TCP's flow control. Each side maintains a "window" that its peer can send data into. When the window is filled up, the peer must stop sending data. As buffer space becomes available, the receiving side informs its peer by sending a SSH_MSG_CHANNEL_WINDOW_ADJUSTMENT message that indicates how many more bytes the peer can send. This message contains two unsigned integers: the *recipient channel* number and a *bytes to add* field that indicates how much to expand the window.

Data is carried in SSH_MSG_CHANNEL_DATA messages. These messages have two fields: the usual *recipient channel* number and the data itself, carried as a string.

Because a channel can carry more than one type of data, an SSH_MSG_CHANNEL_EXTENDED_DATA message (Figure 7.51) carries the other data types. Currently, this message is used only to carry STDERR data in interactive sessions. The *data type* field indicates the type of extended data the message is carrying. The only defined type at this writing is SSH_EXTENDED_DATA_STDERR (1).

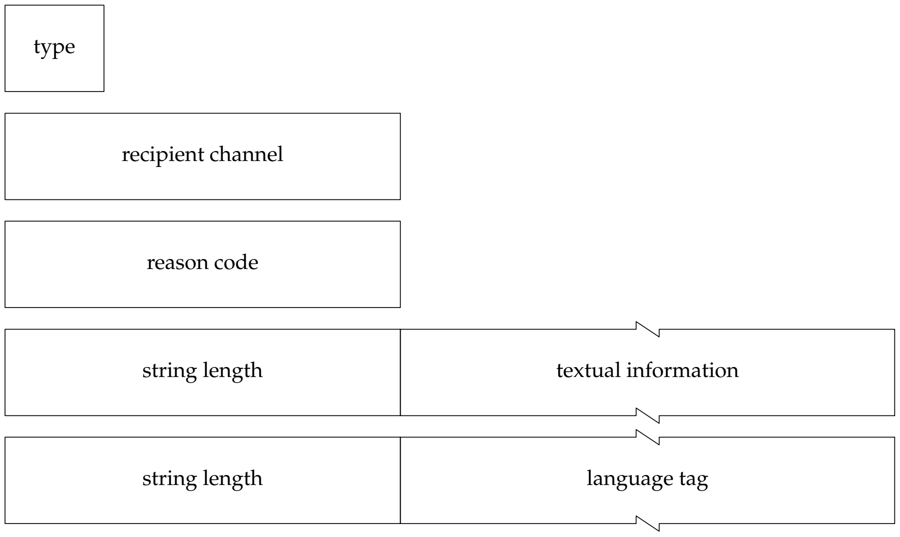

Figure 7.49 The SSH_MSG_CHANNEL_OPEN_FAILURE Message

Code	Value
SSH_OPEN_ADMINISTRATIVELY_PROHIBITED	1
SSH_OPEN_CONNECT_FAILED	2
SSH_OPEN_UNKNOWN_CHANNEL_TYPE	3
SSH_OPEN_RESOURCE_SHORTAGE	4

Figure 7.50 Channel Open Failure Reason Codes

When finished sending data on a channel, a peer can send an SSH_MSG_CHANNEL_EOF message. This message contains a single unsigned integer carrying the recipient's channel number. The message does not close the channel, and the other side may continue to send data—in this respect, it is like the TCP EOF that merely indicates that one side has finished sending data.

Either peer can close the channel by sending an SSH_MSG_CHANNEL_CLOSE message, which again contains only the recipient's channel number. The peer receiving this message must respond with its own SSH_MSG_CHANNEL_CLOSE message, after which no further data can be sent by either side.

Shell and Remote Command Sessions

To start a shell session or execute a remote command, the client opens a session channel by sending the server an SSH_MSG_CHANNEL_OPEN message with the channel type set to "session." If needed, the client can request a pseudo-tty terminal by sending an

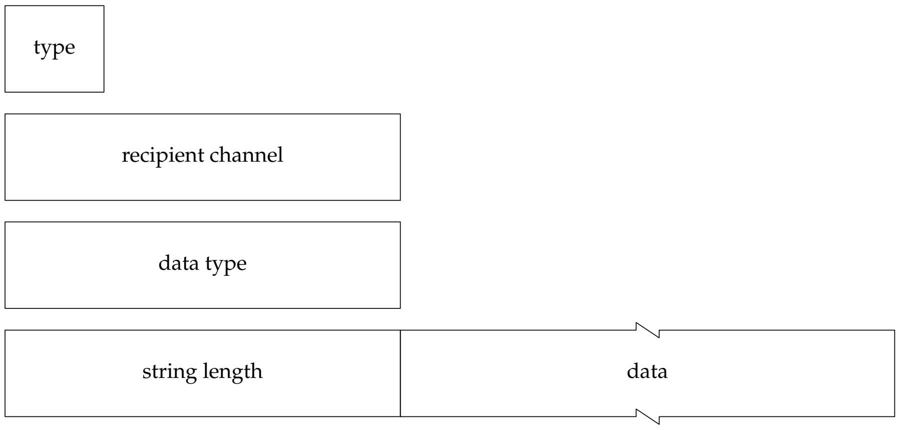

Figure 7.51 The SSH_MSG_CHANNEL_EXTENDED_DATA Message

SSH_MSG_CHANNEL_REQUEST message. The general form of this message is shown in Figure 7.52.

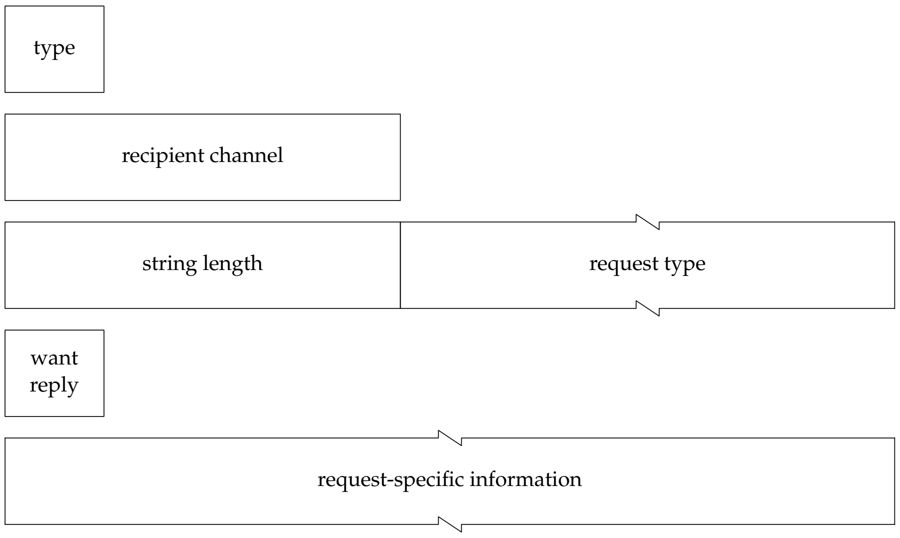

Figure 7.52 The SSH_MSG_CHANNEL_REQUEST Message

To request a pseudo-tty, the *request type* field is set to "pty-req." The additional information shown in Figure 7.53 is also included. This message is almost identical to the SSHv1 SSH_CMSG_REQUEST_PTY message.

If the *want reply* flag is TRUE, the server will reply with either an SSH_MSG_CHANNEL_SUCCESS or an SSH_MSG_CHANNEL_FAILURE message. Both of these messages have a single argument carrying the recipient channel number.

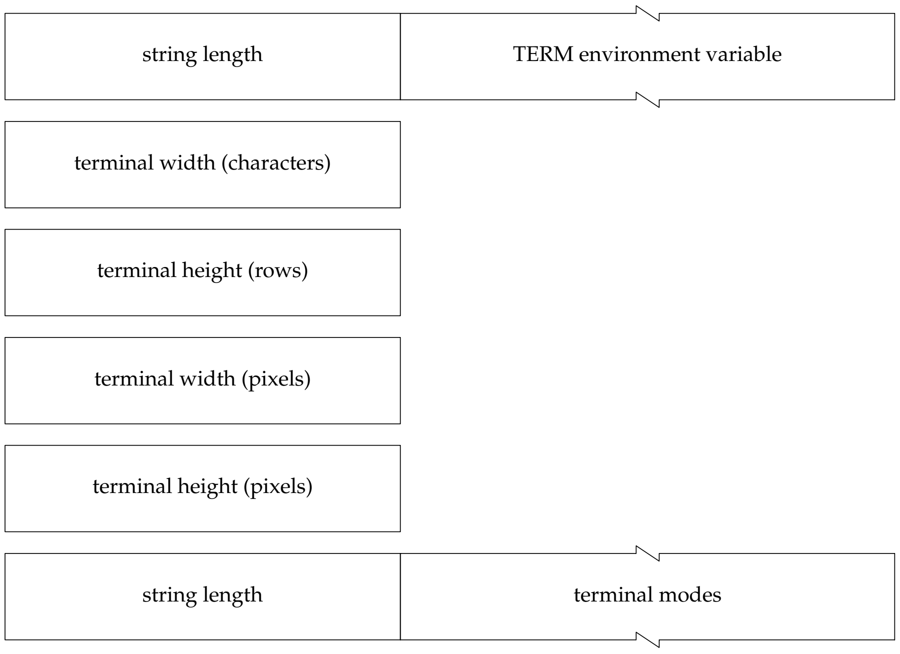

Figure 7.53 Message-Specific Data for a Pseudo-TTY Request

Once the channel is established and a pseudo-tty is started, if needed, the client can request the server to either start a shell or execute a command by sending another SSH_MSG_CHANNEL_REQUEST message. These commands have the *request type* field set to either "shell" or "exec." When the request type is "exec," an additional string parameter carries the name of the command to execute. Before starting the shell or command, the client can set environment variables for them by sending an SSH_MSG_CHANNEL_REQUEST message with the *request type* field set to "env." Two strings in the request-specific data contain the variable name and its value.

When it terminates, the command or shell can return an exit status to the client by sending an SSH_MSG_CHANNEL_REQUEST with the *request type* field set to "exit-status," and an additional unsigned integer containing the exit status. The *want reply* flag must be set to FALSE for this message.

If the shell or command terminates prematurely with a signal, the server sends an SSH_MSG_CHANNEL_REQUEST message with the *request type* field set to "exit-signal." Additional data about the signal is included in the message-specific data, as shown in Figure 7.54. As with the "exit-status" message, the *want reply* field is set to FALSE. The *core dumped* flag is set to TRUE if a core dump resulted and to FALSE otherwise.

The client can send a signal to the command or shell executing on the server by sending the server an SSH_MSG_CHANNEL_REQUEST message with the *request type* field set to "signal." The message-specific data is a string containing the name of the signal, without the "SIG." As with the exit messages, the *want reply* flag is set to FALSE.

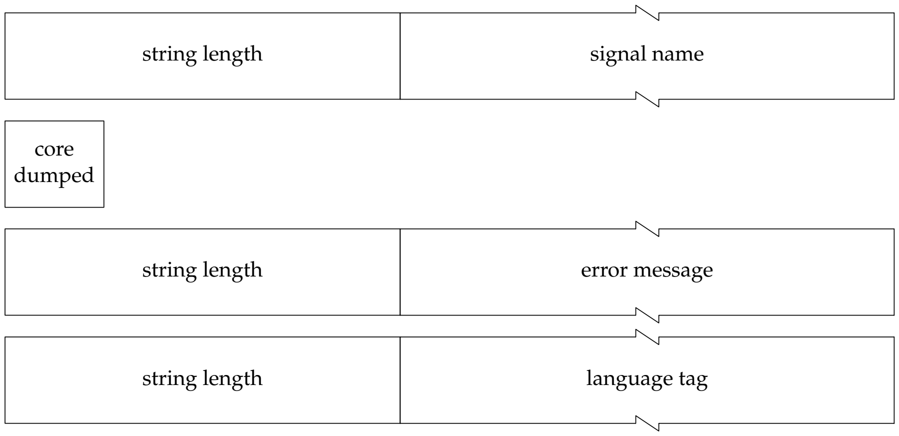

Figure 7.54 Request-Specific Data for the Exited with Signal Message

Not all servers implement signals, of course, so the server will ignore this message if it does not support signals or does not recognize the signal type.

Two other channel messages are associated with interactive sessions. The first is the "window-change" request, which informs the server of a change in the client's terminal window dimensions. The message-specific data comprises four unsigned integers that contain the window's new width and height, first in columns and rows and then in pixels. The *want reply* flag is FALSE.

The other message type associated with interactive sessions is the "xon-xoff" message, which allows the client to do manual (^S/^Q) flow control. The server sends this message to the client to tell it whether the client can do manual flow control. The message-specific data is a single Boolean flag, *can do*, which is TRUE if the client is allowed to perform manual flow control and FALSE otherwise. The *want reply* flag is FALSE for this message.

X11 Forwarding

X11 forwarding under SSHv2 works pretty much the way it does under SSHv1. The client requests forwarding by sending the server an SSH_MSG_CHANNEL_REQUEST message with the request type set to "x11-req." The request-specific information is shown in Figure 7.55. The *single connection* field is a Boolean that indicates whether the server should accept multiple X11 connections. The other fields serve the same purposes they did in SSHv1 X11 forwarding.

When an X11-client connects to the proxy server, the SSH server opens a channel to the client by sending an SSH_MSG_CHANNEL_OPEN message, with the channel type set to "x11." The request-specific data comprises a string containing the originator's address and an unsigned integer containing the originator's source port. As usual, the

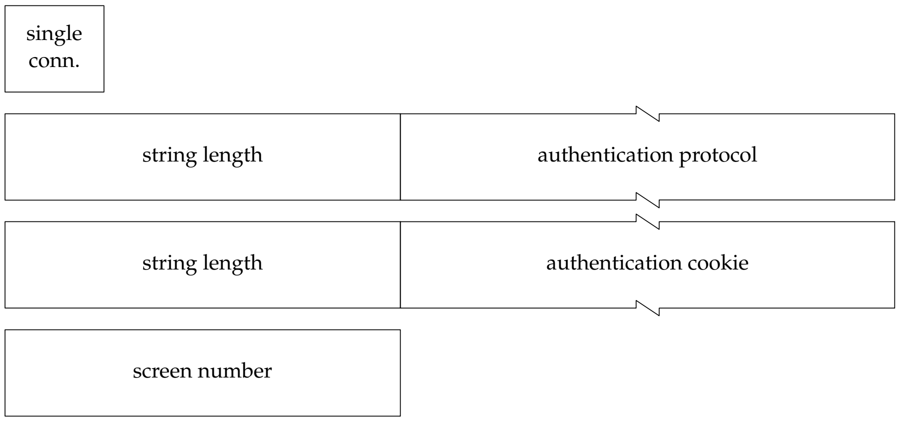

Figure 7.55 Request-Specific Data for the X11 Forwarding Request Message

client is expected to reply with either an SSH_MSG_CHANNEL_OPEN_CONFIRMATION or an SSH_MSG_CHANNEL_OPEN_FAILURE message.

Port Forwarding

As with X11 forwarding, SSHv2 port forwarding is similar to its SSHv1 counterpart. The client may forward a local port to the server at any time by sending the server an SSH_MSG_CHANNEL_OPEN message with the *channel type* field set to "direct-tcpip." The request-specific data, shown in Figure 7.56, indicates the host and port to connect to and the host and port of the connection's originator.

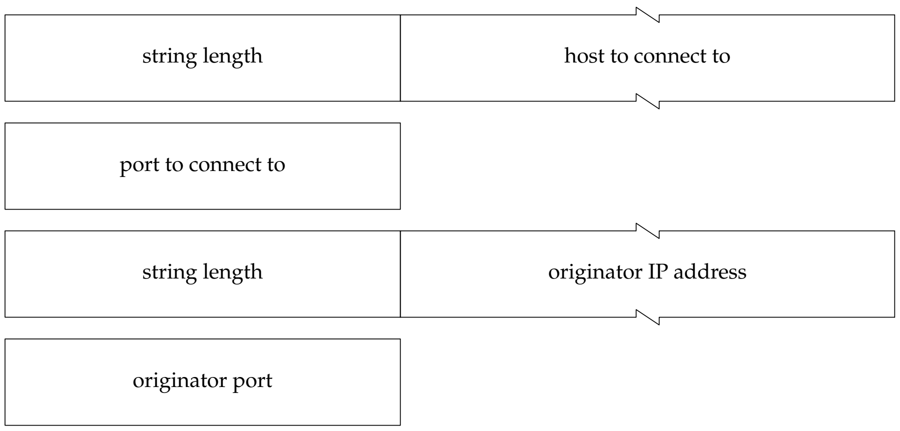

Figure 7.56 Request-Specific Data for a Local Port Forwarding Message

Remote port forwarding is handled just as it is in SSHv1. The client requests the server to forward a port to it by sending an SSH_MSG_GLOBAL_REQUEST message with the request type set to "tcpip-forward." The request-specific data comprises a string containing the address to bind and an unsigned integer containing the port to forward. If the client specifies 0 as the port to forward, the server will bind the next available port number. In this case, if the *want reply* flag is TRUE, the server's SSH_MSG_GLOBAL_REQUEST_SUCCESS message will include an integer indicating the port bound.

When it connects to the server, a remote host will send the client an SSH_MSG_CHANNEL_OPEN message with the *channel type* field set to "forwarded-tcpip." The request-specific data, shown in Figure 7.57, contains the address and port that was connected to and the originator's address or port.

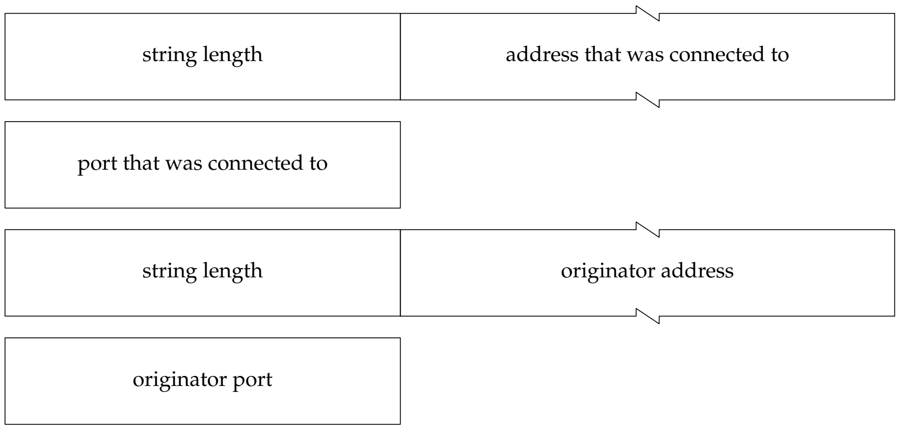

Figure 7.57 Request-Specific Data for a Remote Port Forwarding Message

The client can cancel a forwarding request by sending the server an SSH_MSG_GLOBAL_REQUEST message with a request type of "cancel-tcpip-forward," message-specific data comprising a string with the address that was bound, and an unsigned integer containing the port that was bound.

Subsystems

SSHv2 introduced a new abstraction layer called *subsystems*. A good way of understanding subsystems is to think of them as a sort of `inetd` for SSH. The idea is that the server's system administrator can give arbitrary names to commands that the client may request the server to run.

The most common example is `sftp`, an alternative interface to the `scp` functionality. Under SSHv2, `sftp` is defined as a subsystem. Thus, when the `sftp` client runs, it spawns `ssh` as a subprocess and asks it to run the sftp subsystem on the server. From a functional point of view, this is identical to what happens with SSHv1 except that the name of the process implementing the sftp server need not be "sftp" or even known by the client at all.

The server's system administrator configures the sftp subsystem by providing a mapping between the subsystem name and the name and path of the program implementing it. The system administrator can configure arbitrary subsystems by merely providing a mapping between the subsystem name and the command implementing it. For example, we could implement an rbiff subsystem by mapping the subsystem name "rbiff" to the command `rbiffd`. A more realistic example of SSH subsystems is at `<http://www.columbia.edu/kermit/skermit.html>`, which describes how to run `kermit` as an SSH subsystem.

Pushing our `inetd` analogy a little further, some systems could be implemented internally by `sshd`, much like the `echo` command is implemented internally by most implementations of `inetd.` Currently, no subsystems are implemented in this way, but the capability exists.

The client requests the server to invoke a subsystem by sending the server an SSH_MSG_GLOBAL_REQUEST message with the *request type* field set to "subsystem." The message carries the name of the system in the message-specific information as a string.

Security Issues with SSHv2

As we remarked earlier, SSHv2 was written to address certain security flaws in SSHv1. Although no defects as devastating as the CRC-32 MAC exploit have been discovered in SSHv2, some problems need to be addressed. A simple chosen plaintext attack, usually called Rogaway's attack, against block ciphers in CBC mode is described in [Dai 2002]. The attack requires knowing what the next IV for the block cipher will be, but in CBC mode, one need merely observe the previous cipher block, because it will be used as the next IV.

In Rogaway's attack, the attacker suspects that the value of plaintext block P_i is X and wants to verify that guess. The attacker observes that the last cipher block sent was C_{j-1} and so chooses the plaintext block $P_j = X \oplus C_{i-1} \oplus C_{j-1}$ to be encrypted. Now with CBC mode, we have

$$C_j = \mathrm{E}_K(P_j \oplus C_{j-1}) = \mathrm{E}_K(X \oplus C_{i-1} \oplus C_{j-1} \oplus C_{j-1}) = \mathrm{E}_K(X \oplus C_{i-1})$$

so if $C_j = C_i$, the attacker will have verified that $X = P_i$.

The attack is a little more difficult than indicated, because the first block of a packet starts with the two length fields (40 bits), over which the attacker does not have complete control. This implies that the attacker may have to wait several packets to launch the attack. See [Bellare, Kohno, and Namprempre 2002] for a complete analysis.

Dai recommended using RC4 to avoid this problem. The draft SSH architecture document from the IETF SECSH working group recommends the insertion of SSH_MSG_IGNORE messages to ensure that the attacker will not know the next IV.

Another attack on the SSHv2 transport protocol is described in [Bellare, Kohno, and Namprempre 2002]. The attack is a chosen ciphertext attack that also takes advantage of CBC mode. The idea is that the attacker knows one message M_i and suspects that another message, M_j, is identical or related to it. The attacker can verify this guess by sending the receiver specially crafted ciphertext. If the guess is correct, the receiver will accept the ciphertext as legitimate. If the guess is incorrect, the message will fail the authentication step, and the receiver will disconnect. The details of the attack are in [Bellare, Kohno, and Namprempre 2002].

We can make two observations about these attacks. First, neither approaches the seriousness of the attacks on SSHv1. The most the attacker can do is verify a previous guess as to the value of a block or that two blocks are the same or related. Second, both attacks depend on the underlying encryption using CBC mode.

The second observation suggests that we should consider using something other than a block cipher in CBC mode for encryption. As Dai suggests, RC4 is one possible alternative. [Bellare, Kohno, and Namprempre 2002] considers several other alternatives and concludes that using block encryption in CTR mode is the best solution. The authors have submitted a draft RFC to the SECSH working group, recommending that CTR mode be added to the list of supported block ciphers. The authors also provide tighter bounds on the amount of data that the SSHv2 transport layer can safely send before rekeying.

At the current time, SSHv2 appears to be reasonably secure. The known exploits are difficult to mount and yield relatively small results. These exploits depend on weaknesses in the CBC mode of block ciphers and can therefore be avoided by using RC4 or by using CTR mode instead of CBC mode with block ciphers. The modularity of the SSHv2 architecture and the ability to negotiate the security mechanisms for each session make it easy for SSHv2 to adapt to newly discovered exploits.

As always, we should keep in mind that a secure application can be attacked in many ways, some not directly aimed at the cryptographic mechanisms. For example, an interesting side-channel attack on SSH is described in [Song, Wagner, and Tian 2001]. In this attack, keystroke timings from an interactive SSH session are measured by observing the packet times. Statistical techniques are used to infer information about the text, including in some cases the text typed. One application of these techniques was able to reduce the work in an exhaustive search for passwords by a factor of 50. This attack depends on the fact that in interactive mode, SSH sends each character typed in a separate packet.

7.4 Building VPNs with SSH

We have already seen how SSH can provide a secure tunnel between applications, and in this sense, it might be said to function as a VPN as we've previously defined that term. In this section, we show how to use SSH to join two networks—instead of two applications—with a secure tunnel.

Like its SSL counterpart, this VPN can suffer from performance problems due to the interactions between two instances of TCP. For this reason, the VPNs that we describe are probably not suitable for production environments, but they are easy to set up and so make excellent ad hoc solutions to small problems or as a way of temporarily linking two networks with a VPN.

In Chapter 6, we built our VPN by running PPP through an SSL connection. We could do that here as well in an essentially identical way.

An example of this using Linux is provided in [Kolesnikov and Hatch 2002].

Instead of using PPP, we illustrate another method by making use of a `gtunnel`-based application that we call `sshvpn`.

We set up our tunnel between `bsd` and `laptop`, using the architecture shown in Figure 7.58.

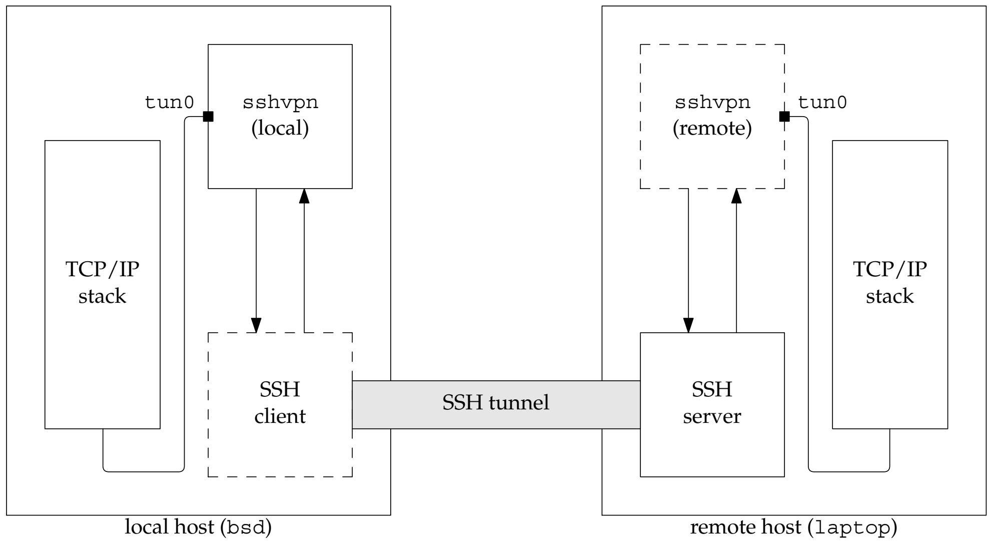

Figure 7.58 Architecture for the SSH VPN

Notice that the figure is similar to Figure 7.22: `sshvpn` starts an SSH client, with which it communicates via pipes. The SSH client, in turn, connects to the SSH server on `laptop`, and the SSH server starts another instance of `sshvpn`, which it communicates with via pipes. As in Figure 7.22, we have shown the two spawned processes as dashed boxes. All communication between the two instances of `sshvpn` travels through the SSH tunnel and so is encrypted and authenticated.

The rest of Figure 7.58 shows that the two instances of `sshvpn` are connected to their hosts' TCP/IP stacks via the `tun0` interface. Thus, data written into the `tun0` interface on `bsd` will be read by the `sshvpn` on `bsd`, sent through the SSH tunnel to the `sshvpn` on `laptop`, and, finally, written into the `tun0` interface on `laptop` by the `sshvpn` running there.

The figure is a little misleading because the traffic that flows through the SSH tunnel must first travel through the TCP/IP stack again. For clarity, we have omitted showing that path.

A convenient way of visualizing this process is to think of the sshvpn/ssh combination on bsd and the sshvpn/sshd combination on laptop as network interfaces that happen to communicate with each other using the SSH protocol. To the applications that use the VPN, the tun0 interface appears to be exactly this. On the one hand, it appears like any other network interface—an Ethernet interface, say—and on the other hand, it communicates with its peer interface over an SSH tunnel.

The sshvpn Application

As with our gtunnel-based IP-in-IP tunnel from Chapter 4, we need specify only startup, inbound, and outbound functions to add to our gtunnel skeleton. Let's begin with startup (Figure 7.59). This function behaves differently, depending on whether it is run on the local (SSH client) side or the remote (SSH server) side. On the local side, it must start ssh and establish the pipes between ssh and itself. On the remote side, sshvpn will be started by sshd, so startup need only map infd and outfd to STDIN and STDOUT. We signal which action startup should perform by calling sshvpn with a single parameter containing our peer's name or address when on the local side and with no parameters on the remote side.

sshvpn.c

```
void startup( int argc, char **argv, int *infd, int *outfd )
{
    int pid;
    int pipein[ 2 ];
    int pipeout[ 2 ];

    if ( argc == 1 )                        /* is spawned sshvpn? */
    {
        *infd = 0;
        *outfd = 1;
        return;
    }
    if ( pipe( pipein ) || pipe( pipeout ) )
        error( 1, errno, "couldn't open pipe" );
    pid = fork();
    if ( pid < 0 )
        error( 1, errno, "couldn't fork" );
    if ( pid )                              /* is parent? */
    {
        close( pipein[ 1 ] );
        close( pipeout[ 0 ] );
        *infd = pipein[ 0 ];
        *outfd = pipeout[ 1 ];
        return;
    }
    dup2( pipeout[ 0 ], 0 );                /* child */
    dup2( pipein[ 1 ], 1 );
    close( pipein[ 0 ] );
    close( pipeout[ 1 ] );
    execl( "/usr/bin/ssh", "ssh", argv[ 1 ],
30         "/home/jcs/bin/sshvpn", NULL );
31     error( 1, errno, "couldn’t exec ssh" );
32 }
```
sshvpn.c

Figure 7.59 The `sshvpn startup` Function

`startup`

6–11 We first test whether the program was started on the local or the remote side. If called with no parameters, the program was spawned by `sshd`, so we need only map STDIN and STDOUT onto `infd` and `outfd`.

12–13 If the program reaches this point, it is operating on the local side, so we must start `ssh`. We start by allocating two pipes, one each for input and output.

14–16 The call to `fork` spawns a child process that is identical to its parent, so there are now two copies of `sshvpn` running. In the parent process, `fork` will return the process ID of the child. In the child process, `fork` will return 0. This return value is the only difference between the parent and child at this point.

17–24 If this is the parent process, we close the write end of the input pipe and the read end of the output pipe, map the pipes onto the `infd` and `outfd` file descriptors, and return to the `main` function.

25–31 In the child process, we map the pipes onto STDIN and STDOUT and close the read end of STDOUT and the write end of STDIN. Notice that the meanings of `pipein` and `pipeout` are reversed in the child—that is, the "in" and "out" are from the perspective of the parent process. Finally, we exec `ssh`. The `execl` call causes the `ssh` image to be read on top of the `sshvpn` process running in the child process. After this call, the child process is running `ssh`, not `sshvpn`. The `execl` call is the same as if we had invoked `ssh` on the command line as

```
ssh remote_host_name /home/jcs/bin/sshvpn
```

If the `execl` call is successful, it will not return (because `ssh` is running instead of `sshvpn`), so it's an error if it does return.

The `outbound` function is shown in Figure 7.60.

sshvpn.c
```
void outbound( int tun, int pipefd )
{
    int rc;
    char buf[ 1500 ];

    rc = read( tun, buf, sizeof( buf ) );
    if ( rc < 0 )
        error( 1, errno, "read returned %d", rc );
    rc = write( pipefd, buf, rc );
    if ( rc < 0 )
        error( 1, errno, "write to pipe returned %d", rc );
}
```
sshvpn.c

Figure 7.60 The `sshvpn outbound` Function

outbound

37–39 The `outbound` function is very simple. It merely reads IP datagrams from the TCP/IP stack and

40–42 writes them into the pipe for delivery to SSH.

The `inbound` function, shown in Figure 7.61, is equally simple.

sshvpn.c

```
44 void inbound( int tun, int pipefd )
45 {
46     int rc;
47     char buf[ 1500 ];

48     rc = read( pipefd, buf, sizeof( buf ) );
49     if ( rc < 0 )
50         error( 1, errno, "read from pipe returned %d", rc );
51     if ( rc == 0 )
52         error( 1, 0, "EOF from peer\n" );
53     rc = write( tun, buf, rc );
54 }
```

sshvpn.c

Figure 7.61 The `sshvpn` `inbound` Function

inbound

48–52 The function reads an IP datagram from SSH. If it gets an EOF from its peer, it terminates `sshvpn`, thus tearing down the VPN.

53 The IP datagram from our peer is sent up the TCP/IP stack by writing it to the tunnel device (`tun0`).

Running `sshvpn`

Running the `sshvpn` program is easy. After configuring the `tun0` devices on `bsd` and `laptop`, we start `sshvpn`:

```
bsd# ./sshvpn laptop
Password:                                   enter laptop's root password
                                            VPN runs until we kill it
^CKilled by signal 2.                       VPN killed
```

The password prompt is from SSH, asking us to authenticate ourselves to the SSH server on `laptop`. After the authentication, `sshd` on `laptop` starts a local copy of `sshvpn`, and the VPN is up and ready to pass traffic. We can test this by `ping`ing `laptop` from `bsd`:

```
$ ping 192.168.2.1
PING 192.168.2.1 (192.168.2.1): 56 data bytes
64 bytes from 192.168.2.1: icmp_seq=0 ttl=64 time=2.999 ms
64 bytes from 192.168.2.1: icmp_seq=1 ttl=64 time=2.600 ms
64 bytes from 192.168.2.1: icmp_seq=2 ttl=64 time=1.791 ms
64 bytes from 192.168.2.1: icmp_seq=3 ttl=64 time=1.741 ms
^C
```

```
--- 192.168.2.1 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max/stddev = 1.741/2.283/2.999/0.536 ms
```

As in the case of our SSL VPN from Chapter 6, it is instructive to consider what the packets traveling through the tunnel look like. In the case of our `ping`s from the preceding example, we have Figure 7.62.

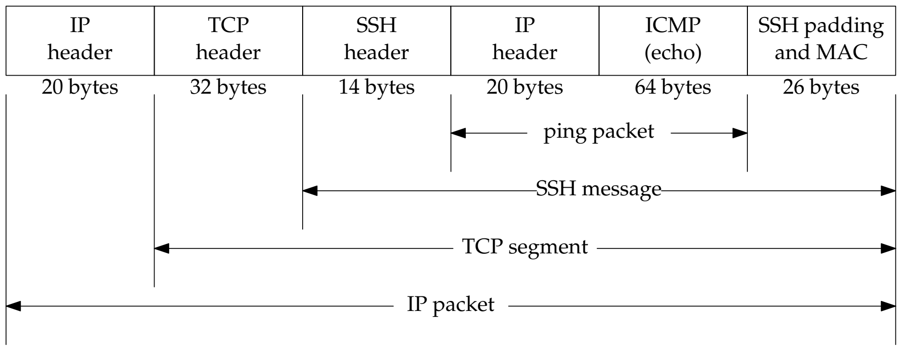

Figure 7.62 SSH VPN Datagram Format

In this case, the `ping` packet is 84 bytes. The entire packet is 176 bytes, so there is a 92-byte overhead (55 percent). This overhead would be less dramatic if the payload packet were larger. For example, the overhead would be only 7 percent for a 1,300-byte payload.

As with all VPNs that use TCP as a transport medium, the more significant overhead is with round-trip time. The average `ping` response was 2.283 milliseconds, which is comparable to what we saw with the SSL VPN in Chapter 6. As before, the `ping` response times over the LAN, without the VPN, are about an order of magnitude less:

```
$ ping laptop
PING laptop.jcs.local (172.30.0.6): 56 data bytes
64 bytes from 172.30.0.6: icmp_seq=0 ttl=64 time=0.252 ms
64 bytes from 172.30.0.6: icmp_seq=1 ttl=64 time=0.258 ms
64 bytes from 172.30.0.6: icmp_seq=2 ttl=64 time=0.357 ms
64 bytes from 172.30.0.6: icmp_seq=3 ttl=64 time=0.262 ms
^C
--- laptop.jcs.local ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.252/0.282/0.357/0.043 ms
```

As mentioned in Chapter 6, the `ping` test is optimal in the sense that the payload is not TCP, so there are not two instances of TCP with competing reliability mechanisms.

SSH VPNs are easy to set up and may make sense if performance is not an issue or if the majority of the traffic will be UDP datagrams. SSH VPNs are especially handy for temporary ad hoc VPNs.

7.5 Summary

In this chapter, we studied the SSH protocol, a drop-in replacement for `telnet`, `ftp`, and the BSD r-commands. The protocol has two incompatible versions, but most implementations support them both.

SSHv1, the older protocol version, can provide a secure remote shell session, a secure channel between any two distributed applications—whether or not they are SSH-aware—X11 and port forwarding, and the secure file copy utility, `scp`. The server and user authentication mechanisms that SSHv1 uses ensure that the SSH client is talking to the correct server, and that the user is authorized to connect to the server.

Finally, SSHv1 has fundamental security problems. The chief problem is the use of a CRC as a MAC for authenticating messages. These problems led to the development of SSHv2.

SSHv2 is a completely different protocol. It replaces the monolithic architecture of SSHv1 with a layered architecture where an SSHv2 transport protocol provides a vehicle for the SSHv2 authentication and connection protocols. The server authentication mechanism is also different from that of SSHv1.

We concluded our examination of SSHv2 by looking at its security properties, and saw that the only attacks are not very serious and can, in any event, be avoided by not using CBC mode while encrypting. We concluded that SSHv2 provides good security and provides a flexible infrastructure that will allow it to adapt to changing conditions.

Finally, we used SSH and our `gtunnel` skeleton to build a VPN between two machines. As with our SSL VPN from Chapter 6, this VPN is not appropriate for high-volume production work but can be useful for ad hoc and temporary solutions to problems requiring a VPN.

Exercises

7.1 In SSHv2 key generation, both the session ID and the exchange hash are included in the key calculation (Figure 7.33). Given that the session ID has the same value as the exchange hash, why are they both included?

7.2 In the SSHv2 public key authentication method, the server does not send the client a challenge, as in the SSHv1 RSA method, but sends a signature over predetermined data (see Figure 7.38). Explain why the SSHv2 method is resistant to replay attacks.

7.3 Show how to build an SSH VPN between two networks by using PPP instead of `gtunnel`.

7.4 In principle, it would be possible to build a hardware implementation of our SSH VPN by embedding a TCP/IP stack and SSH protocol firmware into an Ethernet card. Is this a practical suggestion? Why or why not? Notice how Figure 7.22 becomes an exact depiction of the network architecture if we do this.