CHAPTER

Secure Shell (Remote Access) 25

What You Will Learn

In this chapter, you will learn how the secure shell (SSH) is used as a more secure method of remote access than Telnet. We'll talk about the SSH model, features, and architectures.

You will learn how the SSH protocols operate and how keys are distributed. We'll do a simple example of Diffie-Hellman key distribution using only a pocket calculator and no advanced mathematics.

Not too long ago, most TCP/IP books would routinely cover Telnet as the Internet application for remote access. But today, with the focus on security the Telnet daemon is considered just too dangerous to leave running on hosts and routers, mainly because it is such a tempting target even when password encryption is mandated. There are ways to "enhance" Telnet with security mechanisms, much as the control connection used for FTP (which is little more than a Telnet session) has done.

This is not to say that remote access itself is not an essential Internet and TCP/IP tool. This book could not have been written without Telnet remote access. But more and more today, the preferred application for remote access is SSH.

Windows users should not let the use of the Unix term "shell" scare them. SSH is not really a Unix shell, such as the Bourne shell or other Unix interfaces. It's really a protocol that runs, like most things, over IPv4 or IPv6. Yet the use of the word "shell" in SSH is a good one because there is a lot more to SSH than just remote access. Perhaps the term "secure suite" would have been better, but SSH is what it is.

USING SSH

Most people know SSH as just another way to access the remote host of a router. For example, to access router `CE0` from host `bsdclient` and log in as `admin`, we would use the `-l` option as follows:

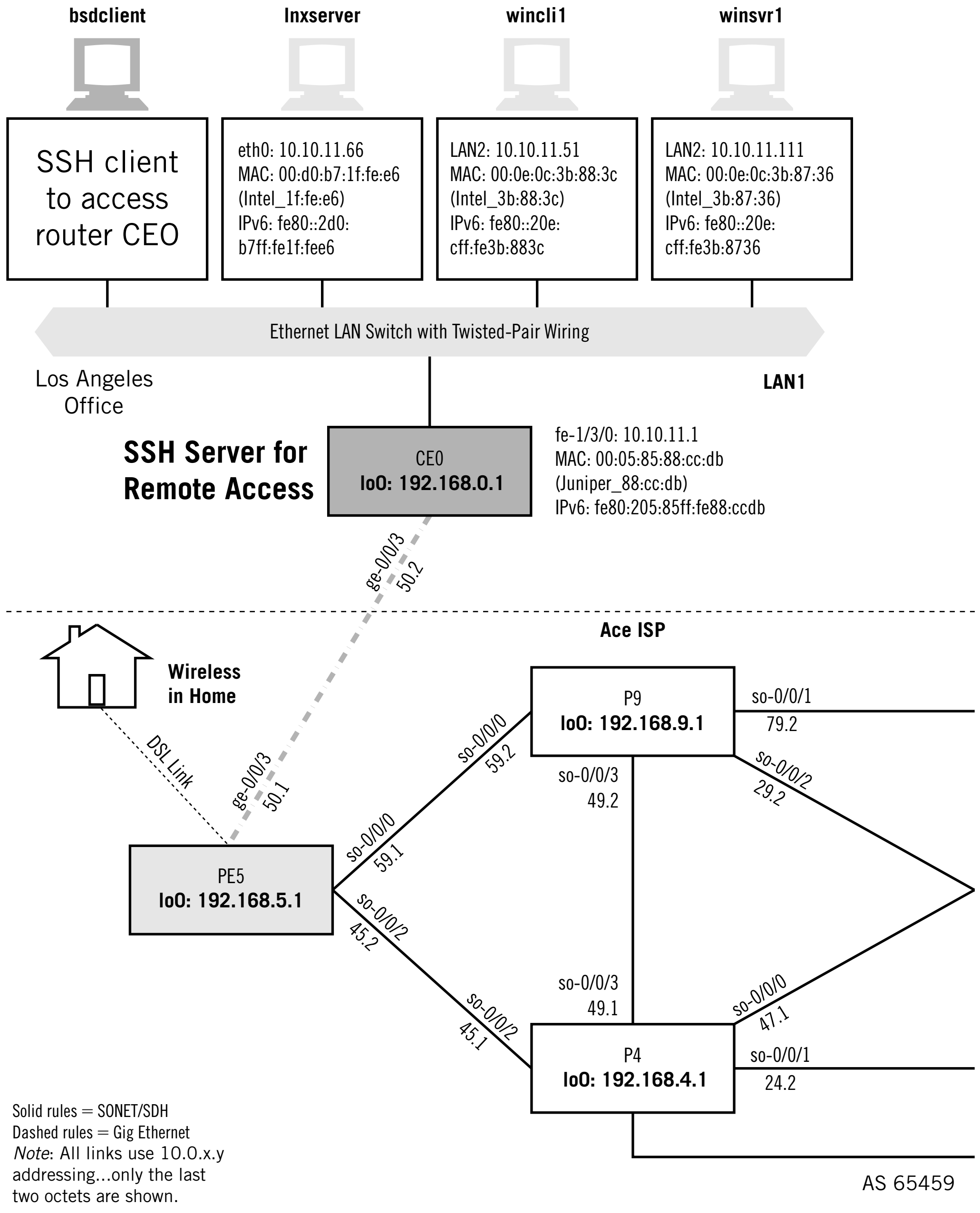
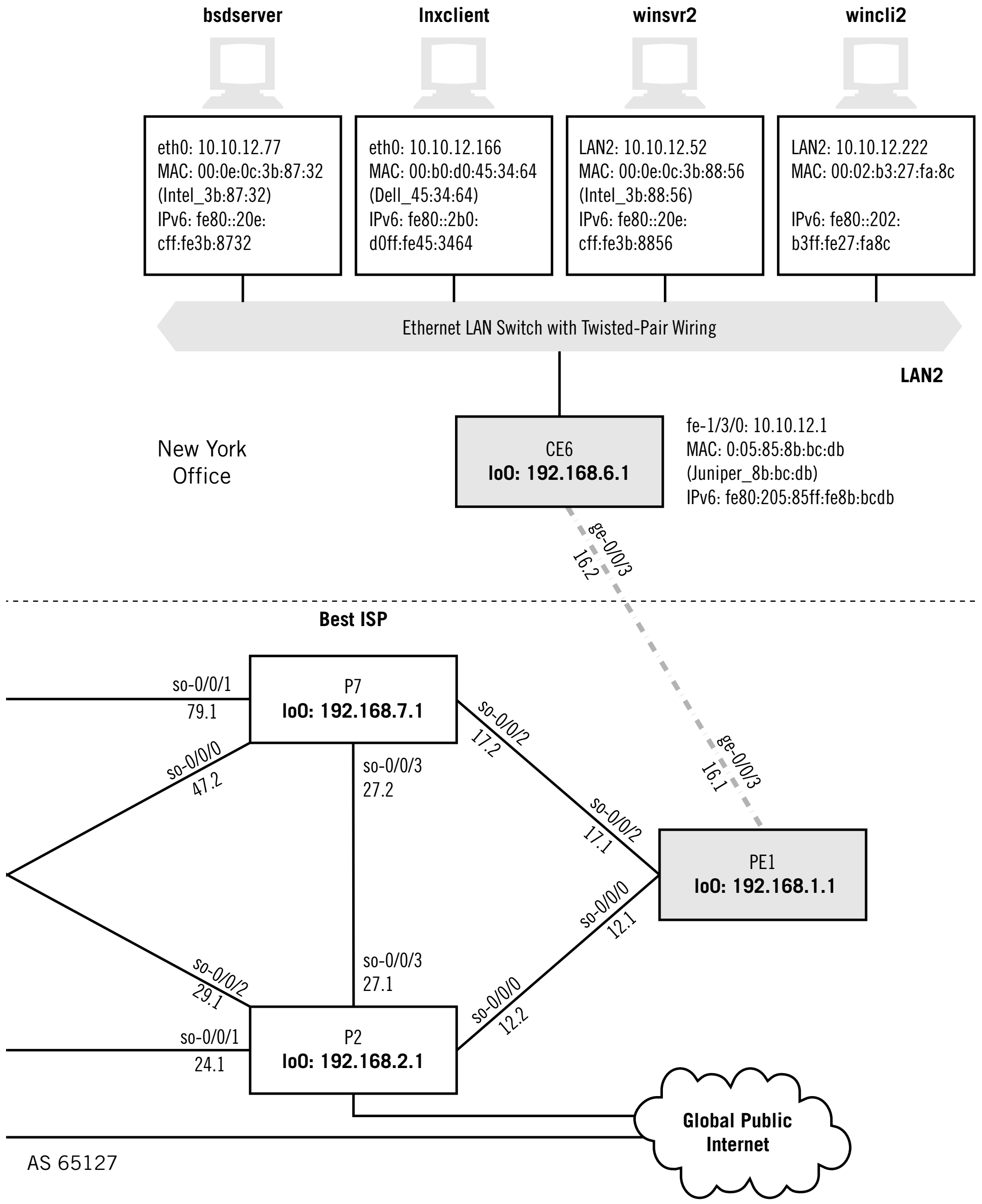


FIGURE 25.1

Using SSH on the Illustrated Network showing the host used as the SSH client and the target router used as the SSH server for remote access.


bsdserver
lnxclient
winsvr2
wincli2
eth0: 10.10.12.77
MAC: 00:0e:0c:3b:87:32
(Intel_3b:87:32)
IPv6: fe80::20e:
cff:fe3b:8732
eth0: 10.10.12.166
MAC: 00:b0:d0:45:34:64
(Dell_45:34:64)
IPv6: fe80::2b0:
d0ff:fe45:3464
LAN2: 10.10.12.52
MAC: 00:0e:0c:3b:88:56
(Intel_3b:88:56)
IPv6: fe80::20e:
cff:fe3b:8856
LAN2: 10.10.12.222
MAC: 00:02:b3:27:fa:8c
IPv6: fe80::202:
b3ff:fe27:fa8c
Ethernet LAN Switch with Twisted-Pair Wiring
LAN2
New York Office
CE6
lo0: 192.168.6.1
fe-1/3/0: 10.10.12.1
MAC: 0:05:85:8b:bc:db
(Juniper_8b:bc:db)
IPv6: fe80:205:85ff:fe8b:bcdb
ge-0/0/3
16.2
Best ISP
so-0/0/1
79.1
P7
lo0: 192.168.7.1
so-0/0/2
17.2
ge-0/0/3
16.1
so-0/0/0
47.2
so-0/0/3
27.2
so-0/0/2
17.1
PE1
lo0: 192.168.1.1
so-0/0/0
12.1
so-0/0/2
29.1
so-0/0/3
27.1
so-0/0/0
12.2
so-0/0/1
24.1
P2
lo0: 192.168.2.1
Global Public Internet
AS 65127

```
bsdclient# ssh -l admin 10.10.11.1

admin@10.10.11.1's password: (not shown)
--- JUNOS 8.4R1.3 built 2007-08-06 06:58:15 UTC

admin@CEO>
```

You might notice a longer wait after issuing the `ssh` command than other commands before being asked for the password, but if the network is fast enough this delay is marginal. In fact, a blizzard of messaging is crisscrossing the network between command and password requests, and even more before the remote device prompt appears. Without some explanation, these messages are completely opaque to users. So, let's use `bsdclient` and `CEO` (as shown in Figure 25.1) to explore SSH a little before looking at the messages in detail.

SSH Basics

Although not technically a shell, SSH lets a user do all of the things Unix commands such as `rsh`, `rlogin`, and `rcp` do. (SSH is sometimes implemented as `slogin`.) SSH is an application that allows users to log on to another host over the network, execute commands on the remote host, and move files around. But unlike the older "r commands" it is intended to replace, SSH provides secure communication over unsecure channels, strong authentication and encryption, and other security features.

The "r commands" were vulnerable to many different types of attacks. Anyone without root access to the hosts or access to the packets on the network can gain unauthorized access to the hosts in several ways. Malicious users can also log all traffic to and from the host, including other users' passwords. (In contrast, SSH *never* sends passwords in clear text.)

The popular X Windows GUI for Unix is also vulnerable in many ways. SSH allows the creation of secure remote X Windows sessions that are transparent to the user. In fact, using SSH for remote X Window clients is easier for users. Users can still use their old `rhosts` and `/etc/hosts` files for this type of remote access, and if a remote host does not support SSH there is a way for the session to fall back to *rsh*.

SSH is a traditional client/server protocol. The SSH server process waits for commands (requests) from SSH clients, executes the command if allowed, and returns the result (reply) to the client. Users are often authenticated with an encrypted key and *passphrase* instead of a password, and these public key files are placed on the remote computers users can access. The overall use of SSH is shown in Figure 25.2.

SSH consists of several client programs and a few configuration files. The programs the user runs are `ssh` or `slogin` (both essentially the same) and `scp` or `sftp` (also the same), depending on implementation. Secure shell keys are managed with `ssh-keygen`, `ssh-agent`, and `ssh-add`.

There have been two major versions of SSH. SSH1 was developed by Tatu Ylonen at the Helsinki University of Technology in Finland in 1995 after a network attack. It was released as free software and source code. It also became an Internet draft, but several

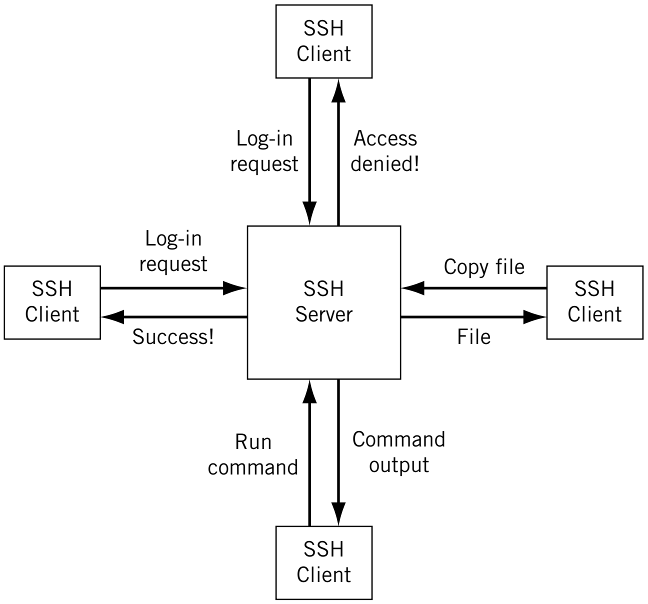


FIGURE 25.2

SSH model. Note that a way to run commands and copy files is included in the model.

issues with the original (which was not systematically developed) were addressed as SSH2 in 1996. SSH2 has new methods and is not compatible with SSH1. Unfortunately, users still liked a lot of the features of SSH1 that were lacking in SSH2, and because some security is better than none, they felt little reason to switch (licensing played a role as well).

OpenSSH is now available as a free implementation of the SSH2 protocol, and it is this version that has been ported to many operating systems. People still talk about the "Ylonen SSH," "SSH1.5," or "OpenSSH" implementations of the basic SSH protocol. SSH was an Internet draft status for a long time, and this chapter describes SSH2. SSH is now defined in a series of RFFCs from RFC 4250 through RFC 4256. This group of RFCs details various aspects of SSH operation.

SSH Features

SSH has excellent protection features. The major ones follow:

Secure client/server communication—All data are encrypted on the network.

Varied authentication—Users can be authenticated by password (encrypted), the host, or a public key.

Authentication integration—SSH can be optionally integrated (and often is) with other authentication systems such as Kerberos, PAM, PGP, and SecureID.

Security add-on—SSH can be used to add security to applications such as NNTP, Telnet, VNC, and a lot of other TCP/IP protocols and applications.

Transparency and versatility—SSH can be transparent to the user and there are implementations for almost all operating systems (including Windows with OpenSSH implementations).

SSH protects users against:

IP spoofing—A remote host can send IP packets pretending to come from somewhere else, such as a trusted host. Spoofers on LANs can even pretend to be the local routers to the outside world, which SSH protects against as well.

IP source routing—This is another way for hackers to claim that a packet came from another host.

DNS spoofing—Hackers can forge name server records supplied to a host.

Intermediate device control—This is an old favorite. A hacker can take control of a router or host between hosts and execute many types of data manipulation.

Clear text interception—Data or passwords sent in clear text are always targets for hackers.

X Windows attacks—Hackers can listen to X Windows authentication exchanges and spoof server connections.

SSH *never* trusts the network. Even if hackers took over the entire network, all that can happen is that SSH is forced to disconnect. Hackers cannot decrypt, play back, or compromise data on the connection.

This is not to say that the SSH is perfect. Like any other tool, SSH is only as good as those setting it up and using it. For example, SSH does have an option for encryption type (`none`), but this is only to be used for testing purposes. (There is no real enforcement of this, of course.) And SSH does nothing to prevent someone who had gained access to the host another way (perhaps by sitting down in front of the unprotected host itself) from doing a lot of damage with root access. In that case, SSH is often the first target of a local hacker.

In addition, a lot of organizations with their own firewall devices are nervous when users rely on SSH to connect to hosts. Remember, everything in the SSH stream is encrypted, and fairly well at that. What SSH does is offer users a direct pipeline to their internal machines right through the firewall, an invisible tunnel into the organization.

There are ways to work around this through a SSH proxy gateway, including the "mute shell" and "SSH-in-SSH" approaches. But nothing is ever perfect or 100% secure.

SSH Architecture

Many SSH components interact to allow secure client-server exchanges. These components, not all of which are distinct programs or processes, are shown in Figure 25.3.

The following is a *brief* overview of the major components of SSH.

Server—The program that authenticates and authorizes SSH connections, usually `sshd`.

Client—The program run on the client (user) device, often `ssh`, but also `scp`, `sftp`, and so on.

Session—The client/server connection, which can be interactive or batch. The session begins after successful authentication to the server and ends when the connection terminates.

Key generator—A program (usually `ssh-keygen`) that generates persistent keys. (Key types are discussed later in this chapter.)

Known hosts—A database of host keys. This is the major authentication mechanism in SSH.

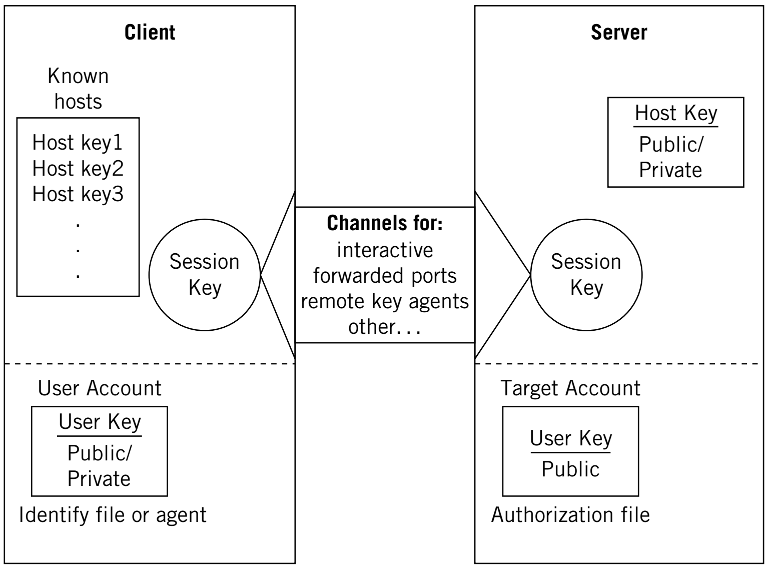


FIGURE 25.3

An overview of the SSH architecture. Note that a lot of space is devoted to the distribution and use of encryption keys.

Agent—A caching program for user keys to spare users the need to repeat passphrases. The agent is only a convenience and does not disclose the keys. The usual agent is `ssh-agent`, and `ssh-add` loads and unloads the key cache.

Signer—This program signs the host-based authentication packets used instead of password authentication.

Random seed—Random data used by SSH components to initialize the pseudo-random number generators (PRNG) used in SSH.

Configuration files—Settings to determine the behavior of SSH clients and servers.

SSH Keys

Keys are a crucial part of SSH. Almost everything that SSH does involves a key, and often more than one key. SSH keys can range from tens of bits to almost 2000. Keys are used as parameters for SSH algorithms such as encryption or authentication. SSH keys are used to bind the operation to a particular user.

There are two types of SSH keys: symmetric (shared secret keys) and asymmetric (public and private key pairs). As in all public key systems, asymmetric keys are used to establish and exchange short-duration symmetric keys. The three types of keys used in SSH are outlined in Table 25.1. As mentioned, user and host keys are typically created by the `ssh-keygen` program.

User key—This persistent asymmetric key is used by the SSH clients to validate the user's identity. A single user can have multiple keys and "identities" on a network.

Host key—This persistent asymmetric key is used by the SSH servers to validate their identity, as well as the client if host-based authentication is used. If the device runs a single SSH server process, the host key uniquely identifies the device. Devices running multiple SSH servers can share a key or use different host keys.

Session key—This transient symmetric key is generated to encrypt the data sent between client and server. It is shared during the SSH connection setup to use

Table 25.1 SSH Key Name Types and Major Characteristics

Key Name	Lifetime	Creator	Type	Purpose
User key	Persistent	User	Public	Identify user to server
Host key	Persistent	Administrator	Public	Identify a server or device
Session key	One session	Client and server	Secret	Secure communications

for encrypted data streams during the session. When the session ends, the key is destroyed. There are several session keys, actually—one in each direction and others to check integrity of communications.

SSH Protocol Operation

This section describes the operations of SSH2 and not the older, and incompatible, SSH1. There are four major pieces to SSH, and they are documented separately and theoretically have nothing whatsoever to do with one another. In practice, they all function together to provide the set of features and functions that make up SSH. Each is still an Internet draft, but these should all become RFCs some day.

There are some other documents that extend these four protocols, but these make up the heart of SSH. The major protocols follow:

- SSH Transport Layer Protocol (SSH-TRANS)
- SSH Authentication Protocol (SSH-AUTH)
- SSH Connection Protocol (SSH-CONN)
- SSH File Transfer Protocol (SSH-SFTP)

The relationships between the protocols, and their major functions, are shown in Figure 25.4.

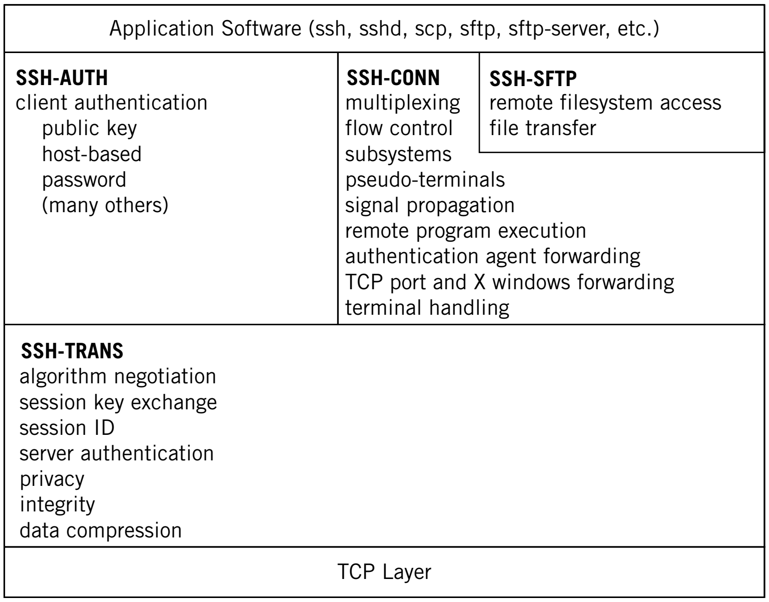


FIGURE 25.4

SSH protocols, showing how they relate to one another and the TCP transport layer.

All critical parameters used in all of the protocols are negotiated. These parameters include the ways and algorithms used for:

- User authentication
- Server authentication
- Session key exchange
- Data integrity and privacy
- Data compression

In most categories, clients and servers are required to support one or more methods, thereby promoting interoperability. Support is not the same as implementation, however, and specific clients and servers still have to find a "match" to accomplish their goals.

Initial connections (including server authentication, basic encryption, and integrity services) are established with SSH-TRANS, which is the fundamental piece of SSH. An SSH-TRANS connection provides a single and secure data stream operating full-duplex between client and server.

Once the SSH-TRANS connection is made, the client can use SSH-AUTH for authentication to the server. Multiple authentication methods can be used, and SSH-AUTH establishes things such as the format and order of requests, conditions of success or failure, and so on. Protocol extensions are defined to allow the methods to be extended in the future as other authentication methods are developed. Only one method is *required* in SSH-AUTH: public key using the digital signature standard (DSS). Two more methods are *defined*: password and host-based (but we'll concentrate on public key in this chapter).

Once authenticated, SSH clients use the SSH-CONN protocol over the "pipe" established by SSH-TRANS. There are multiple interactive or batch (noninteractive) sessions over SSH *channels*. The sessions include things such as X Windows and TCP forwarding (tunneling), control signaling (such as `^C`) over the connection, data compression, and related activities.

If file transfer or remote file manipulation is needed, this is provided by the SSH-SFTP protocol. The sequence of invoking these protocols is not rigid, and there is considerable variation in implementation, mostly in "nonstandard" or customized environments where global client access is neither needed nor desired.

Note that the SSH protocols only define what should happen on the network. Internals such as how keys are stored on the local disk, user authorization, and key forwarding (which most people think of as intimate parts of SSH), are really implementation-dependent pieces that are usually completely incompatible. The following sections describe some of the key aspects of protocol operation.

Transport Layer Protocol

Clients normally access the SSH process on the server at well-known TCP port 22. The server announces the SSH version in a text string, and there are certain conventions built into this string. For example, SSH version "1.99" means that the server supports both SSH1 and SSH2, and the client can choose to use either one from then on. Of course, if the client and server are not compatible, either can break the connection at that point.

If the connection goes forward, SSH-TRANS shifts into the *binary packet protocol*—a record-oriented non-text protocol defined for SSH-TRANS. The first activity here is *key exchange*, which precedes the negotiation of the basic security properties of the SSH session.

The key exchange often employs some form of the *Diffie-Hellman* procedure for key agreement, although there are others. Diffie-Hellman describes a way to securely exchange information (such as a shared secret key) over an unsecured network such as the Internet by using asymmetric public/private keys established beforehand. The key exchange itself should be authenticated to guard against "man-in-the-middle" attacks.

Pocket Calculator Diffie-Hellman

In the SSL chapter, we did an exercise in "pocket calculator public key encryption" to show that although the mathematical theory behind the use of asymmetric public/private key encryption was complex its use was not. We've mentioned Diffie-Hellman several times, and when first popularized in 1976 Diffie-Hellman was so revolutionary some doubted it actually worked (not mathematicians, of course!). How could secure shared secret keys possibly be sent over an unsecure network where anyone can make copies of the packets?

Let's show how Diffie-Hellman can be used to allow users to share a secret key and yet no one else knows what the key is (even the "man-in-the-middle" vulnerability does not really "crack" the key, just hijacks it). Again, we'll use small non–real-world numbers just to make the math easy enough to do on a pocket calculator. We've already shown how to raise the numbers to a power, and to compute the modular remainder from division, so that is not repeated.

Like public key encryption, Diffie-Hellman depends on properties of prime numbers. There are two important ones: the very large prime itself (P) and a related number (derived by formula) called the "primitive root of P," which is usually called Q. A large prime P will have many primitive roots, but only one is used. For this example, let's use $P = 13$ and $Q = 11$ (I didn't use a formula: There are tables on primes and primitive roots all over the Internet).

According to usual security example practice, let's call our two correspondents Alice (A) and Bob (B). A and B exchange these two numbers publicly over the network, without worrying if anyone else knows them (they have no choice, because the network is by definition unsecure anyway).

A and B each pick, independently, a random number (naturally, in reality this is done by software without users "picking" anything). Let's use $A = 4$ and $B = 7$ (they can even pick the same number by chance, of course). Now each calculates A^* and B^* according to the following formulas:

- A computes $A^* = Q^A \bmod (P) = 11^4 \bmod (13) = 14{,}641 \bmod 13 = 3$
- B computes $B^* = Q^B \bmod (P) = 11^7 \bmod (13) = 19{,}487{,}171 \bmod 13 = 2$

Now, all A and B have to do is exchange their A* and B* numbers over the network—not caring who sees them (which they can't help anyway). But wait, couldn't someone easily figure out the A and B values in the example? Yes, of course, with the small numbers used here. But when large enough primes and well-chosen primitive roots are selected, and A and B choose random enough numbers (one reason you don't let A and B pick their own numbers), there are many numbers that give the values 3 and 2.

Now A and B simply calculate the shared secret key to use:

- A's secret key = $(B^*)^A$ mod (P) = 2^4 mod (13) = 16 mod 13 = 3
- B's secret key = $(A^*)^B$ mod (P) = 3^7 mod (13) = 2187 mod 13 = 3

Given enough time, the shared secret key can be broken. So, the Diffie-Hellman process is repeated constantly (at fixed intervals), recomputing new keys, sometimes every few seconds. By the time the key is broken, a new one is in use.

The key exchange is usually repeated during a session because "stale" keys that are used too long might allow a malicious user to break the encryption that much faster. The more often the keys are changed the less likely this becomes, and even if broken only that portion of the session is compromised. Usually SSH key exchanges occur every hour or after every gigabyte of data.

The use of the "null" cipher, which means no encryption at all, is a valid choice for SSH clients and servers, but this is only to be used for testing. However, many SSH administrators never disable it. A favorite OpenSSH trick is to gain root access to a host and edit the user's configuration file (`~/.ssh/config`) so that all hosts use the null cipher only. If client or server do *not* support "null," this evil trick is not possible.

Key exchange and encryption choice are followed by more security parameter choices. Methods of integrity, server authentication, and compression (a marginal feature still considered part of SSH security) are agreed on. Public key systems are popular choices, but the issue is always how to verify proper ownership of the public key, as discussed in Chapter 23, where certificates were introduced as a way to provide server authentication. At the end of the process, methods for cipher/integrity/compression are established for client-to-server and server-to-client exchanges.

Authentication Protocol

SSH-AUTH is simpler than SSH-TRANS. The authentication protocol defines a framework for these exchanges, defines a number of actual mechanisms (but only a few of them), and allows for extensions. The three defined methods are public-key, password, and host-based authentication.

The authentication process is framed by client requests and server responses. The "authentication" request actually includes elements of authorization (access rights are checked as well). A request contains:

Username, U—The claimed identity of the user. On Unix systems, this is typically the user account. However, the interpretation context is not defined by the protocol.

Server name, S—The user is requesting access to a "server," which is really the protocol to run on the SSH-TRANS connection after authentication finishes. This is usually "ssh-connection," which represents all services (remote log-in, command execution, etc.) provided by the SSH-CONN protocol.

Method name, M, and method-specific data, D—The particular authentication method used for the request and any data needed with it. For example, if the method is `password`, the data provided are the password itself.

There can be other messages exchanged, depending on the authentication request. But ultimately the server issues an authentication response. The response can be `SUCCESS` or `FAILURE`, and the success message has no other content. The failure response includes

- a list of the authentication methods that can continue the process
- a "partial success" flag

The `FAILURE` response can be misleading. If the partial success flag is not set (false), the message means that the preceding authentication method has failed for some reason (incorrect password, invalid account, and so on). However, if the partial success flag is set (true), the message means that the method has *succeeded* (odd in a failure message!), but the server requires that additional methods also succeed before access is granted. In other words, the server can require multiple successful authentication methods. OpenSSH does not support this feature.

But how does the client know which methods to start with? The client starts with a "none" authentication request, which prompts the server to reply with a list of the authentication methods the client can choose to continue the process. In other words, if the server requires any authentication at all, the "none" method fails. If not, a `SUCCESS` is immediate and a lot of time is saved.

The Connection Protocol

Clients usually request to use "ssh-connection" after a successful authentication exchange. Once the server starts the service, SSH uses the SSH-CONN protocol. This is really when SSH starts to do things.

The basic SSH-CONN service is multiplexing: the creation of dynamic logical *channels* over the SSH-TRANS connection. Channels are identified by numbers and can be created and destroyed by either side of the connection. Channels are flow controlled and have a *type*, which are also extensible. The defined channels types follow:

Session—These are for the remote execution of a program. Opening a channel does not start a program, but when started several session channels can be in operation at once.

x11—These channels are for X Windows operations.

forwarded-tcpip—These inbound channels are for forwarded TCP ports. *(Port forwarding* in SSH just means that SSH transparently encrypts and decrypts data on a TCP port.) The server opens this channel type back to the client to carry remotely forwarded TCP port data.

direct-tcpip—These outbound TCP channels are used to connect to a socket. The client simply starts listening on the port indicated.

SSH-CONN defines a set of channel or global requests in addition to traditional channel operations such as open, close, send, and so on. The global requests follow:

tcpip-forward—Used to request remote TCP port forwarding. This feature is not yet supported by Open SSH.

cancel-tcpip-forward—Used to cancel remote TCP port forwarding.

The channel requests are more elaborate and are only summarized in the following. Most refer to the remote side of the session channel.

pty-req—Requests a pseudo-terminal for the channel (usually for interactive applications). Includes window size and terminal mode information.

x11-req—Requests X Window forwarding.

Env—Sets an environmental variable. This can be risky, so it is carefully controlled.

shell, exec, subsystem—Run the default shell for the account, a program, or service. This connects the channel to the standard input and output and error streams. A "subsystem" is used, for example, with file transfers, and the subsystem name is `SFTP` in this case.

window-change—Changes the terminal window size.

xon-xoff—Uses client ^S/^Q flow control.

Signal—Sends a signal (such as the Unix *kill* command) to the remote side.

exit-status—Returns the program's exit status.

exit-signal—Returns the signal that terminated the program.

Although these channel requests can technically be sent from server to client, the use of SSH as a remote access tool means that most of these requests are issued by the client and expect the server to perform in a certain way. Clients usually ignore these requests from a server, just for security reasons.

The File Transfer Protocol

The last piece of the SSH protocol "suite" is SSH-SFTP. Oddly, SSH-SFTP does not really implement any file transfers at all because it has no file transfer capability. What the protocol does is to use SSH to start a remote file transfer agent and then work with it over the secure connection.

Initially, SSH used a secure version of the remote copy (rcp) Unix program to implement secure copy (scp). As rcp ran the remote shell (rsh), so scp ran the secure shell (SSH). But rcp was a very limited program compared to FTP. A session only transferred a group of files in one direction, and it did not allow directory listings, browsing, or any of the other features associated with FTP.

Thus, SSH2 eventually incorporated the idea of SFTP to secure the file transfer process. The SSH-SFTP protocol describes how this happens. Unfortunately, SFTP isn't just using SSH to connect to a remote FTP server. SFTP has absolutely nothing to do with the FTP protocol described in an earlier chapter of this book.

SSH and FTP are not a good match, one reason being that separate connections are used in FTP for control and data transfer. FTP itself (like Telnet) can be made more secure with SSL, but few FTP servers provide these functions. So, an FTP server can also be an SSH server (providing files in unsecure and secure manners)—and that's about a close as SSH and FTP can get.

How does SSH-SFTP work? Well, there are really two ways to transfer files over an SSH connection: with *scp* or with *sftp* (the names might be different, but it's the procedure that's important).

When a client uses *scp*, the transfer begins by running *ssh* with certain options, such as when a forwarding agent is in use. This process in turn runs another version on the remote host, which is, of course, running *sshd*. That copy of *scp* is run with its own (undocumented) options, such as "to" (-t) and "from" (-f). SSH then uses *scp*, now running on client and server, to transfer the file over the secure SSH connection.

Figure 25.5 shows how SSH uses *scp* to transfer a file called `mywebpage.html` to a server and rename it `index.html`. Naturally, the transfer is encrypted and secure.

SSH can even do a trick that FTP does not allow. SSH can be used for "third-party" transfers, a capability never implemented in FTP beyond the testing phase (for security reasons). In other words, when run locally, SSH can transfer a file between two remote hosts (as long as the authentication succeeds).

Consequently, users can perform the Web page transfer to the server even if the page is on their office desktop and they are sitting with a laptop at an airport gate waiting for a flight.

```
scp lnxclient:mywebpage.html lnxserver:index.html
```

Using *sftp* is similar, but the syntax and options for the command are different. This method starts an SSH subsystem, and that means that the SSH server must be specifically configured to run the SFTP protocol. Figure 25.6 shows how the same file

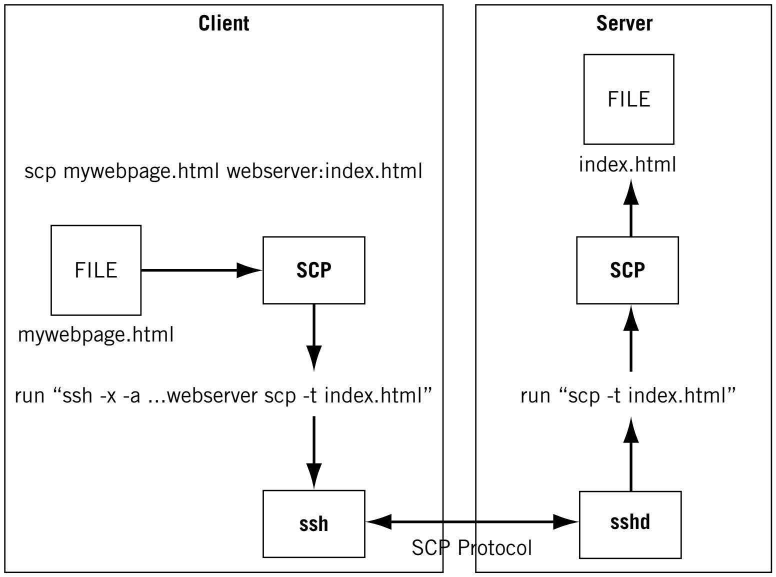


FIGURE 25.5

Transferring files with SCP, showing how SSH is used with the file copy.

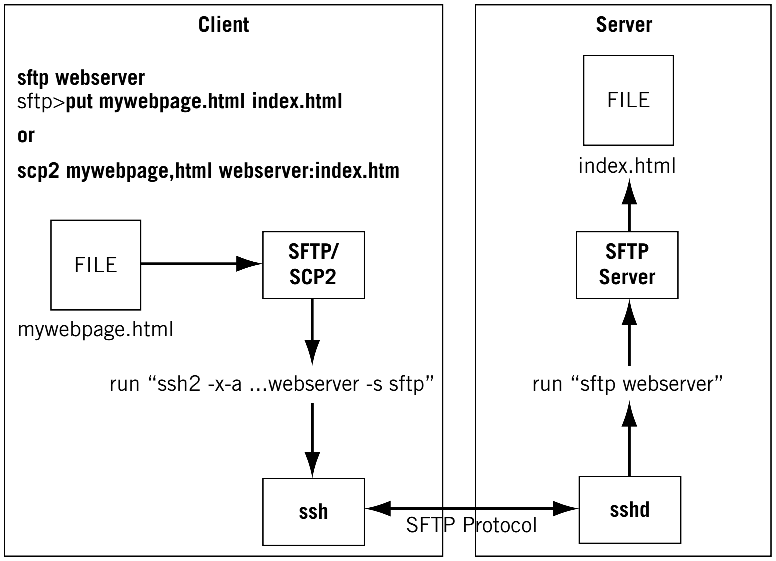


FIGURE 25.6

A file transfer with SFTP, showing the same results as when using SCP.

transfer would be done with *sftp* (in the SSH implementation known as Tectia, *sftp* is confusingly invoked with the command *scp2*).

The point here is that both methods will transfer the file as long as everything else is set up correctly. The best book on SSH—*SSH: The Secure Shell,* by Daniel J. Barrett, Richard E. Silverman, and Robert G. Byrnes (O'Reilly Media)—is about as long as this one. Interested readers are referred to this text for more detailed information on SSH.

SSH IN ACTION

If there is one thing that was used more than FTP to produce this book, it's SSH. In fact, all of the file transfers used to consolidate output for these examples could just as easily have been done with SCP or SFTP. This is especially true when routers are the remote systems: Only in special circumstances will organizations allow or use Telnet for router access.

Let's use SSH to contact the routers on the Illustrated Network. Naturally, the routers have been set up ahead of time to allow administrator access from certain hosts on LAN1 and LAN2 and are running *sshd*. But on the client side, we'll run *ssh* "out of the box" and see what happens.

Ethereal captures are not the best way to look at SSH in action. The secure and encrypted transfers make packet analysis difficult (and often impossible). Fortunately, we can use the debug feature of SSH itself to analyze the exchange in very verbose form (using the `-vv` option).

Let's see if we can catch SSH-TRANS, SSH-AUTH, and SSH-CONN in action when we access router TP2 (`10.10.11.1`) from `bsdclient`. We'll log in (the `-l` option) as `admin`.

```
bsdclient# ssh -vv -l admin 10.10.11.1
OpenSSH_3.5p1 FreeBSD-20030924, SSH protocols 1.5/2.0, OpenSSL 0x0090704f
debug1: Reading configuration data /etc/ssh/ssh_config
debug1: Rhosts Authentication disabled, originating port will not be trusted.
debug1: ssh_connect: needpriv 0
debug1: Connecting to 10.10.11.1 [10.10.11.1] port 22.
debug1: Connection established.
debug1: identity file /root/.ssh/identity type -1
debug1: identity file /root/.ssh/id_rsa type -1
debug1: identity file /root/.ssh/id_dsa type -1
debug1: Remote protocol version 1.99, remote software version OpenSSH_3.8
debug1: match: OpenSSH_3.8 pat OpenSSH*
debug1: Enabling compatibility mode for protocol 2.0
debug1: Local version string SSH-2.0-OpenSSH_3.5p1 FreeBSD-20030924
debug1: SSH2_MSG_KEXINIT sent
debug1: SSH2_MSG_KEXINIT received
debug2: kex_parse_kexinit: diffie-hellman-group-exchange-sha1,diffie-
  hellman-
  group1-sha1
debug2: kex_parse_kexinit: ssh-dss,ssh-rsa
debug2: kex_parse_kexinit: aes128-cbc,3des-cbc,blowfish-cbc,cast128-cbc,
  arcfour,aes192-cbc,aes256-cbc,rijndael-cbc@lysator.liu.se
```

```
debug2: kex_parse_kexinit: aes128-cbc,3des-cbc,blowfish-cbc,cast128-cbc,
 arcfour,aes192-cbc,aes256-cbc,rijndael-cbc@lysator.liu.se
debug2: kex_parse_kexinit: hmac-md5,hmac-sha1,hmac-ripemd160,hmac-ripemd160@
 openssh.com,hmac-sha1-96,hmac-md5-96
debug2: kex_parse_kexinit: hmac-md5,hmac-sha1,hmac-ripemd160,hmac-ripemd160@
 openssh.com,hmac-sha1-96,hmac-md5-96
debug2: kex_parse_kexinit: none,zlib
debug2: kex_parse_kexinit: none,zlib
debug2: kex_parse_kexinit:
debug2: kex_parse_kexinit:
debug2: kex_parse_kexinit: first_kex_follows 0
debug2: kex_parse_kexinit: reserved 0
debug2: kex_parse_kexinit: diffie-hellman-group-exchange-sha1,diffie-
 hellman-
 group1-sha1
debug2: kex_parse_kexinit: ssh-rsa,ssh-dss
debug2: kex_parse_kexinit: aes128-cbc,3des-cbc,blowfish-cbc,cast128-cbc,
 arcfour,aes192-cbc,aes256-cbc,rijndael-cbc@lysator.liu.se,aes128-
 ctr,aes192-ctr,aes256-ctr
debug2: kex_parse_kexinit: aes128-cbc,3des-cbc,blowfish-cbc,cast128-
 cbc,arcfour,aes192-cbc,aes256-cbc,rijndael-cbc@lysator.liu.se,aes128-
 ctr,aes192-ctr,aes256-ctr
debug2: kex_parse_kexinit: hmac-md5,hmac-sha1,hmac-ripemd160,hmac-ripemd160@
 openssh.com,hmac-sha1-96,hmac-md5-96
debug2: kex_parse_kexinit: hmac-md5,hmac-sha1,hmac-ripemd160,hmac-ripemd160@
 openssh.com,hmac-sha1-96,hmac-md5-96
debug2: kex_parse_kexinit: none,zlib
debug2: kex_parse_kexinit: none,zlib
debug2: kex_parse_kexinit:
debug2: kex_parse_kexinit:
debug2: kex_parse_kexinit: first_kex_follows 0
debug2: kex_parse_kexinit: reserved 0
debug2: mac_init: found hmac-md5
debug1: kex: server->client aes128-cbc hmac-md5 none
debug2: mac_init: found hmac-md5
debug1: kex: client->server aes128-cbc hmac-md5 none
debug1: SSH2_MSG_KEX_DH_GEX_REQUEST sent
debug1: expecting SSH2_MSG_KEX_DH_GEX_GROUP
debug1: dh_gen_key: priv key bits set: 136/256
debug1: bits set: 1042/2049
debug1: SSH2_MSG_KEX_DH_GEX_INIT sent
debug1: expecting SSH2_MSG_KEX_DH_GEX_REPLY
debug1: Host '10.10.11.1' is known and matches the DSA host key.
debug1: Found key in /root/.ssh/known_hosts:1
debug1: bits set: 1049/2049
debug1: ssh_dss_verify: signature correct
debug1: kex_derive_keys
debug1: newkeys: mode 1
debug1: SSH2_MSG_NEWKEYS sent
debug1: waiting for SSH2_MSG_NEWKEYS
debug1: newkeys: mode 0
debug1: SSH2_MSG_NEWKEYS received
```

```
debug1: done: ssh_kex2.
debug1: send SSH2_MSG_SERVICE_REQUEST
debug1: service_accept: ssh-userauth
debug1: got SSH2_MSG_SERVICE_ACCEPT
debug1: authentications that can continue: publickey,password,keyboard-
 interactive
debug1: next auth method to try is publickey
debug1: try privkey: /root/.ssh/identity
debug1: try privkey: /root/.ssh/id_rsa
debug1: try privkey: /root/.ssh/id_dsa
debug2: we did not send a packet, disable method
debug1: next auth method to try is keyboard-interactive
debug2: userauth_kbdint
debug2: we sent a keyboard-interactive packet, wait for reply
debug1: authentications that can continue: publickey,password,keyboard-
 interactive
debug2: we did not send a packet, disable method
debug1: next auth method to try is password
admin@10.10.11.1's password: (not shown)
debug2: we sent a password packet, wait for reply
debug1: ssh-userauth2 successful: method password
debug1: channel 0: new [client-session]
debug1: send channel open 0
debug1: Entering interactive session.
debug2: callback start
debug1: ssh_session2_setup: id 0
debug1: channel request 0: pty-req
debug1: channel request 0: shell
debug1: fd 3 setting TCP_NODELAY
debug2: callback done
debug1: channel 0: open confirm rwindow 0 rmax 32768
debug2: channel 0: rcvd adjust 131072
--- JUNOS 8.4R1.3 built 2007-08-06 06:58:15 UTC
admin@CE0>
```

The substantial output captures all three phases of SSH protocol operation (all but SSH-SFTP). Let's see what the major portions of this listing are saying.

Roughly speaking, the first half of the output is SSH-TRANS negotiation to establish the methods to use for key exchange, and what to use for cipher, integrity, and compression. The next quarter is used for SSH-AUTH to decide on a user authentication method to be used (its password). The last quarter, after the password is entered, is SSH-CONN (setting up SSH channel 0 from router to client).

It's not necessary to parse this line by line. Generally, the exchange starts by parsing the version string supplied by the router and starting the negotiation. The router announces support for SSH1 or SSH2 (version 1.99).

```
debug1: Remote protocol version 1.99, remote software version OpenSSH_3.8
debug1: match: OpenSSH_3.8 pat OpenSSH*
debug1: Enabling compatibility mode for protocol 2.0
```

The client announces OpenSSH support as well.

```
debug1: Local version string SSH-2.0-OpenSSH_3.5p1 FreeBSD-20030924
```

Now the process shifts to binary packet mode and begins in earnest. The next major section presents the router and client support set for key exchange, cipher, integrity, and compression.

```
debug1: SSH2_MSG_KEXINIT sent
debug1: SSH2_MSG_KEXINIT received
debug2: kex_parse_kexinit: diffie-hellman-group-exchange-sha1,diffie-
  hellman-group1-sha1
debug2: kex_parse_kexinit: ssh-dss,ssh-rsa
debug2: kex_parse_kexinit: aes128-cbc,3des-cbc,blowfish-cbc,cast128-
  cbc,arcfour,aes192-cbc,aes256-cbc,rijndael-cbc@lysator.liu.se
debug2: kex_parse_kexinit: aes128-cbc,3des-cbc,blowfish-cbc,cast128-
  cbc,arcfour,aes192-cbc,aes256-cbc,rijndael-cbc@lysator.liu.se
debug2: kex_parse_kexinit: hmac-md5,hmac-sha1,hmac-ripemd160,hmac-ripemd160@
  openssh.com,hmac-sha1-96,hmac-md5-96
debug2: kex_parse_kexinit: hmac-md5,hmac-sha1,hmac-ripemd160,hmac-ripemd160@
  openssh.com,hmac-sha1-96,hmac-md5-96
debug2: kex_parse_kexinit: none,zlib
debug2: kex_parse_kexinit: none,zlib
```

The first two lines exchange the messages, which are parsed in pairs in the following. The first pair establishes the key exchange algorithms that the client understands (`diffie-hellman-group-exchange-sha1,diffie-hellman-group1-sha1`), and the second establishes the key types (ssh-dss, ssh-rsa). The other three pairs show that the client and server both support the same methods in the other three categories. (It's not unusual for servers to support methods more than clients.) A long section of back-and-forth negotiation takes place to pare down the possibilities, and finally the client and server agree on what three methods to use for cipher, integrity, and compression.

```
debug1: kex: server->client aes128-cbc hmac-md5 none
debug1: kex: client->server aes128-cbc hmac-md5 none
```

Still, in SSH-TRANS, the actual key exchange and server authentication now begin. Fortunately, it's really the correct router.

```
debug1: SSH2_MSG_KEX_DH_GEX_REQUEST sent
debug1: expecting SSH2_MSG_KEX_DH_GEX_GROUP
debug1: dh_gen_key: priv key bits set: 136/256
debug1: bits set: 1042/2049
debug1: SSH2_MSG_KEX_DH_GEX_INIT sent
debug1: expecting SSH2_MSG_KEX_DH_GEX_REPLY
debug1: Host '10.10.11.1' is known and matches the DSA host key.
debug1: Found key in /root/.ssh/known_hosts:1
debug1: bits set: 1049/2049
debug1: ssh_dss_verify: signature correct
```

The router is known because we've accessed it before (many times, in fact). If we go somewhere we've never been before, we have the option to break off the session because the server cannot be authenticated.

```
debug1: SSH2_MSG_KEX_DH_GEX_REQUEST sent
debug1: expecting SSH2_MSG_KEX_DH_GEX_GROUP
debug1: dh_gen_key: priv key bits set: 145/256
debug1: bits set: 1006/2049
debug1: SSH2_MSG_KEX_DH_GEX_INIT sent
debug1: expecting SSH2_MSG_KEX_DH_GEX_REPLY
debug2: no key of type 0 for host 10.10.12.1
debug2: no key of type 1 for host 10.10.12.1
The authenticity of host '10.10.12.1 (10.10.12.1)' can't be established.
DSA key fingerprint is 51:5f:da:41:41:9d:b1:c0:3f:a7:d0:a8:b9:7c:99:aa.
Are you sure you want to continue connecting (yes/no)?
```

At last we're finished with SSH-TRANS. Now SSH-AUTH is used to authenticate the "user account" to the server. We derive some new keys for the process, and finally (because nothing else "works") allow the user to type in a password for the router.

```
debug1: kex_derive_keys
debug1: newkeys: mode 1
debug1: SSH2_MSG_NEWKEYS sent
debug1: waiting for SSH2_MSG_NEWKEYS
debug1: newkeys: mode 0
debug1: SSH2_MSG_NEWKEYS received
debug1: done: ssh_kex2.
debug1: send SSH2_MSG_SERVICE_REQUEST
debug1: service_accept: ssh-userauth
debug1: got SSH2_MSG_SERVICE_ACCEPT
debug1: authentications that can continue: publickey,password,keyboard-
  interactive
debug1: next auth method to try is publickey
debug1: try privkey: /root/.ssh/identity
debug1: try privkey: /root/.ssh/id_rsa
debug1: try privkey: /root/.ssh/id_dsa
debug2: we did not send a packet, disable method
debug1: next auth method to try is keyboard-interactive
debug2: userauth_kbdint
debug2: we sent a keyboard-interactive packet, wait for reply
debug1: authentications that can continue: publickey,password,keyboard-
  interactive
debug2: we did not send a packet, disable method
debug1: next auth method to try is password
admin@10.10.11.1's password:
```

Although it is difficult to tell from the debug messages, there is a significant wait after the password is typed in while SSH-CONN sets up channel 0 over the SSH-TRANS connection. But finally we're in an interactive session and all set to go.

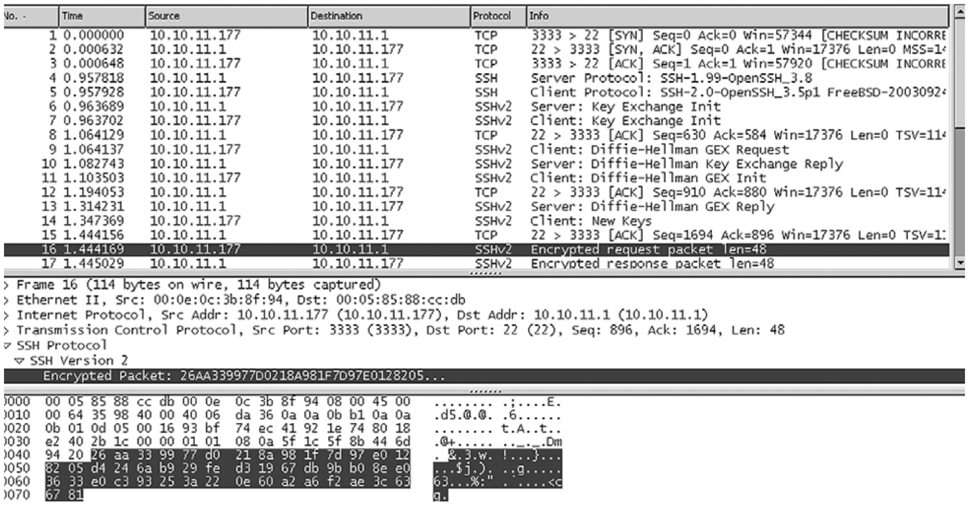
```
debug2: we sent a password packet, wait for reply
debug1: ssh-userauth2 successful: method password
debug1: channel 0: new [client-session]
debug1: send channel open 0
debug1: Entering interactive session.
debug2: callback start
debug1: ssh_session2_setup: id 0
debug1: channel request 0: pty-req
debug1: channel request 0: shell
debug1: fd 3 setting TCP_NODELAY
debug2: callback done
debug1: channel 0: open confirm rwindow 0 rmax 32768
debug2: channel 0: rcvd adjust 131072
--- JUNOS 8.4R1.3 built 2007-08-06 06:58:15 UTC
admin@CEO>
```

Note that SSH does not bypass the router's own authentication method (log-in ID and password) in any way. But it *does* ensure that what the user types in is not sent in plain text over the network.

Let's quickly show `sftp` in action to fetch a file called `tp2` from the router. This shows obvious similarities with FTP use, but is much more secure.

```
bsdclient# sftp admin@10.10.11.1
Connecting to 10.10.11.1...
admin@10.10.11.1's password: (not shown)
sftp> ls
.
..
.ssh
CEO-base
mw-graceful-restart
richard-ASP-manual-SA
richard-base
tp2
wjg-ORA-base
wjg-bgp-try
wjg-ipv6-mcast
wjg-with-ipv6
sftp> get tp2
Fetching /var/home/remote/tp2 to tp2
sftp> quit
bsdclient#
```

The SSH debug sequence for Linux is almost identical to the one for FreeBSD, and also uses OpenSSH. Although not used here, OpenSSH for Windows XP exists and is called PuTTY.



```
No. -  Time      Source          Destination     Protocol  Info
     1 0.000000  10.10.11.177    10.10.11.1      TCP       3333 > 22 [SYN] Seq=0 Ack=0 Win=57344 [CHECKSUM INCORRE
     2 0.000632  10.10.11.1      10.10.11.177    TCP       22 > 3333 [SYN, ACK] Seq=0 Ack=1 Win=17376 Len=0 MSS=1
     3 0.000648  10.10.11.177    10.10.11.1      TCP       3333 > 22 [ACK] Seq=1 Ack=1 Win=57920 [CHECKSUM INCORRE
     4 0.957818  10.10.11.1      10.10.11.177    SSH       Server Protocol: SSH-1.99-OpenSSH_3.8
     5 0.957928  10.10.11.177    10.10.11.1      SSH       Client Protocol: SSH-2.0-OpenSSH_3.5p1 FreeBSD-2003092
     6 0.963689  10.10.11.1      10.10.11.177    SSHv2     Server: Key Exchange Init
     7 0.963702  10.10.11.177    10.10.11.1      SSHv2     Client: Key Exchange Init
     8 1.064129  10.10.11.1      10.10.11.177    TCP       22 > 3333 [ACK] Seq=630 Ack=584 Win=17376 Len=0 TSV=11
     9 1.064137  10.10.11.177    10.10.11.1      SSHv2     Client: Diffie-Hellman GEX Request
    10 1.082743  10.10.11.1      10.10.11.177    SSHv2     Server: Diffie-Hellman Key Exchange Reply
    11 1.103503  10.10.11.177    10.10.11.1      SSHv2     Client: Diffie-Hellman GEX Init
    12 1.194053  10.10.11.1      10.10.11.177    TCP       22 > 3333 [ACK] Seq=910 Ack=880 Win=17376 Len=0 TSV=11
    13 1.314231  10.10.11.1      10.10.11.177    SSHv2     Server: Diffie-Hellman GEX Reply
    14 1.347369  10.10.11.177    10.10.11.1      SSHv2     Client: New Keys
    15 1.444156  10.10.11.1      10.10.11.177    TCP       22 > 3333 [ACK] Seq=1694 Ack=896 Win=17376 Len=0 TSV=1
    16 1.444169  10.10.11.177    10.10.11.1      SSHv2     Encrypted request packet len=48
    17 1.445029  10.10.11.1      10.10.11.177    SSHv2     Encrypted response packet len=48

> Frame 16 (114 bytes on wire, 114 bytes captured)
> Ethernet II, Src: 00:0e:0c:3b:8f:94, Dst: 00:05:85:88:cc:db
> Internet Protocol, Src Addr: 10.10.11.177 (10.10.11.177), Dst Addr: 10.10.11.1 (10.10.11.1)
> Transmission Control Protocol, Src Port: 3333 (3333), Dst Port: 22 (22), Seq: 896, Ack: 1694, Len: 48
▽ SSH Protocol
  ▽ SSH Version 2
      Encrypted Packet: 26AA339977D0218A981F7D97E0128205...

0000  00 05 85 88 cc db 00 0e  0c 3b 8f 94 08 00 45 00   ........ .;....E.
0010  00 64 35 98 40 00 40 06  da 36 0a 0a 0b b1 0a 0a   .d5.@.@. .6......
0020  0b 01 0d 05 00 16 93 bf  74 ec 41 92 1e 74 80 18   ........ t.A..t..
0030  e2 40 2b 1c 00 00 01 01  08 0a 5f 1c 5f 8b 44 6d   .@+..... .._._.Dm
0040  94 20 26 aa 33 99 77 d0  21 8a 98 1f 7d 97 e0 12   . &.3.w. !...}...
0050  82 05 d4 24 6a b9 29 fe  d3 19 67 db 9b b0 8e e0   ...$j.). ..g.....
0060  36 33 e0 c3 93 25 3a 22  0e 60 a2 a6 f2 ae 3c 63   63...%:" .`....<c
0070  67 81                                              g.
```

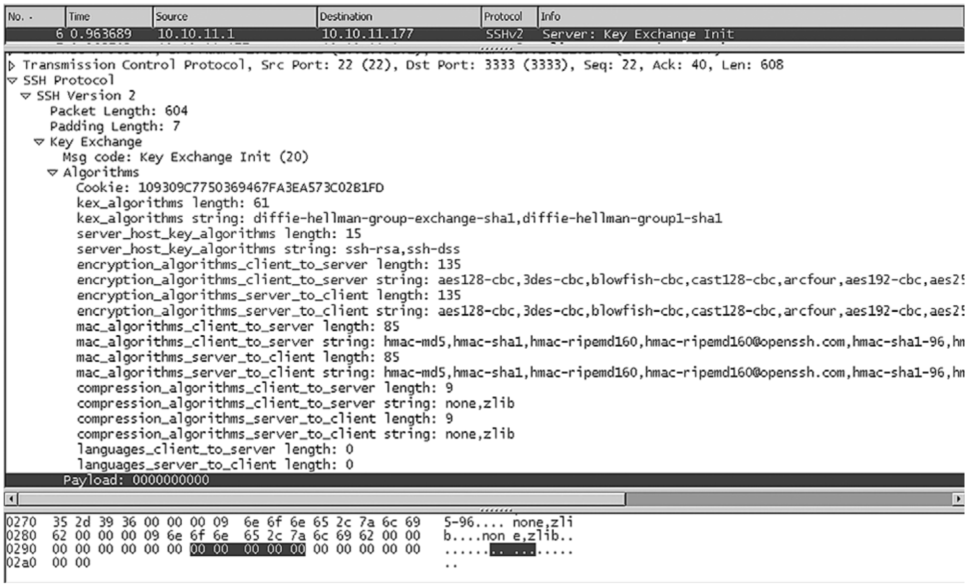
FIGURE 25.7

SSH capture with Ethereal, showing how the packet content is encrypted and therefore not parsed by the utility.

What does SSH look like "on the wire"? Figure 25.7 shows what Ethereal sees at the start of SSH-TRANS, including a look at an encrypted packet.

QUESTIONS FOR READERS

Figure 25.8 shows some of the concepts discussed in this chapter and can be used to answer the following questions.



```
No. .  Time       Source         Destination     Protocol  Info
     6 0.963689   10.10.11.1     10.10.11.177    SSHv2     Server: Key Exchange Init

▷ Transmission Control Protocol, Src Port: 22 (22), Dst Port: 3333 (3333), Seq: 22, Ack: 40, Len: 608
▽ SSH Protocol
  ▽ SSH Version 2
      Packet Length: 604
      Padding Length: 7
    ▽ Key Exchange
        Msg code: Key Exchange Init (20)
      ▽ Algorithms
          Cookie: 109309C7750369467FA3EA573C02B1FD
          kex_algorithms length: 61
          kex_algorithms string: diffie-hellman-group-exchange-sha1,diffie-hellman-group1-sha1
          server_host_key_algorithms length: 15
          server_host_key_algorithms string: ssh-rsa,ssh-dss
          encryption_algorithms_client_to_server length: 135
          encryption_algorithms_client_to_server string: aes128-cbc,3des-cbc,blowfish-cbc,cast128-cbc,arcfour,aes192-cbc,aes2
          encryption_algorithms_server_to_client length: 135
          encryption_algorithms_server_to_client string: aes128-cbc,3des-cbc,blowfish-cbc,cast128-cbc,arcfour,aes192-cbc,aes2
          mac_algorithms_client_to_server length: 85
          mac_algorithms_client_to_server string: hmac-md5,hmac-sha1,hmac-ripemd160,hmac-ripemd160@openssh.com,hmac-sha1-96,hm
          mac_algorithms_server_to_client length: 85
          mac_algorithms_server_to_client string: hmac-md5,hmac-sha1,hmac-ripemd160,hmac-ripemd160@openssh.com,hmac-sha1-96,hm
          compression_algorithms_client_to_server length: 9
          compression_algorithms_client_to_server string: none,zlib
          compression_algorithms_server_to_client length: 9
          compression_algorithms_server_to_client string: none,zlib
          languages_client_to_server length: 0
          languages_server_to_client length: 0
        Payload: 0000000000

0270  35 2d 39 36 00 00 00 09  6e 6f 6e 65 2c 7a 6c 69   5-96.... none,zli
0280  62 00 00 00 09 6e 6f 6e  65 2c 7a 6c 69 62 00 00   b....non e,zlib..
0290  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
02a0  00 00                                              ..
```

FIGURE 25.8

SSH capture with Ethereal.

1. Which devices are communicating here? Is this message from the server to the client or in the opposite direction?
2. Which ports are used on the devices? Is one the usual SSH server port?
3. Which version of SSL is used? What type of message is parsed in the figure?
4. Which two server host key algorithms are supported?
5. How many compression algorithms are supported?