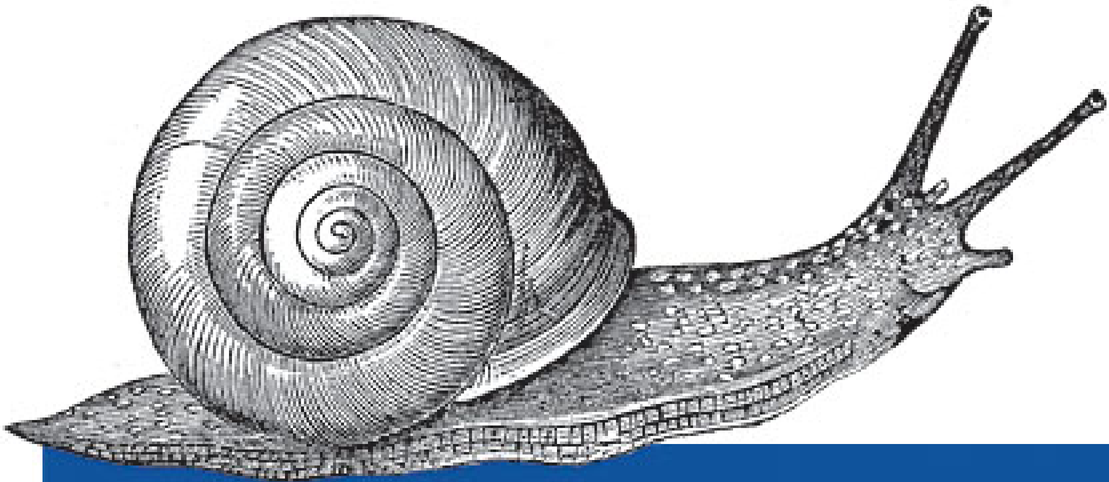
Securing your Network and Services

2nd Edition
Covers SSH-2 Protocol



SSH The Secure Shell

The Definitive Guide

Daniel J. Barrett,
Richard Silverman & Robert G. Byrnes

O'REILLY®

SSH, the Secure Shell: The Definitive Guide™
by Daniel J. Barrett, Richard E. Silverman, and Robert G. Byrnes

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Editor:	Mike Loukides
Production Editor:	Mary Brady
Cover Designer:	Ellie Volckhausen
Interior Designer:	David Futato

Printing History:

February 2001:	First Edition.
May 2005:	Second Edition.

This book uses RepKover™, a durable and flexible lay-flat binding.

ISBN: 0-596-00895-3
[M]

CHAPTER 1

Introduction to SSH

Many people today have multiple computer accounts. If you're a reasonably savvy user, you might have a personal account with an Internet service provider (ISP), a work account on your employer's local network, and a few computers at home. You might also have permission to use other accounts owned by family members or friends.

If you have multiple accounts, it's natural to want to make connections between them. For instance, you might want to copy files between computers over a network, log into one account remotely from another, or transmit commands to a remote computer for execution. Various programs exist for these purposes, such as *ftp* for file transfers, *telnet* for remote logins, and *rsh* for remote execution of commands.

Unfortunately, many of these network-related programs have a fundamental problem: they lack security. If you transmit a sensitive file via the Internet, an intruder can potentially intercept and read the data. Even worse, if you log onto another computer remotely using a program such as *telnet*, your username and password can be intercepted as they travel over the network. Yikes!

How can these serious problems be prevented? You can use an *encryption program* to scramble your data into a secret code nobody else can read. You can install a *firewall,* a device that shields portions of a computer network from intruders, and keep all your communications behind it. Or you can use a wide range of other solutions, alone or combined, with varying complexity and cost.

1.1 What Is SSH?

SSH, the Secure Shell, is a popular, powerful, software-based approach to network security.[*] Whenever data is sent by a computer to the network, SSH automatically encrypts (scrambles) it. Then, when the data reaches its intended recipient, SSH

[*] "SSH" is pronounced by spelling it aloud: S-S-H.

automatically decrypts (unscrambles) it. The result is *transparent* encryption: users can work normally, unaware that their communications are safely encrypted on the network. In addition, SSH uses modern, secure encryption algorithms and is effective enough to be found within mission-critical applications at major corporations.

SSH has a client/server architecture, as shown in Figure 1-1. An SSH *server* program, typically installed and run by a system administrator, accepts or rejects incoming connections to its host computer. Users then run SSH *client* programs, typically on other computers, to make requests of the SSH server, such as “Please log me in,” “Please send me a file,” or “Please execute this command.” All communications between clients and servers are securely encrypted and protected from modification.

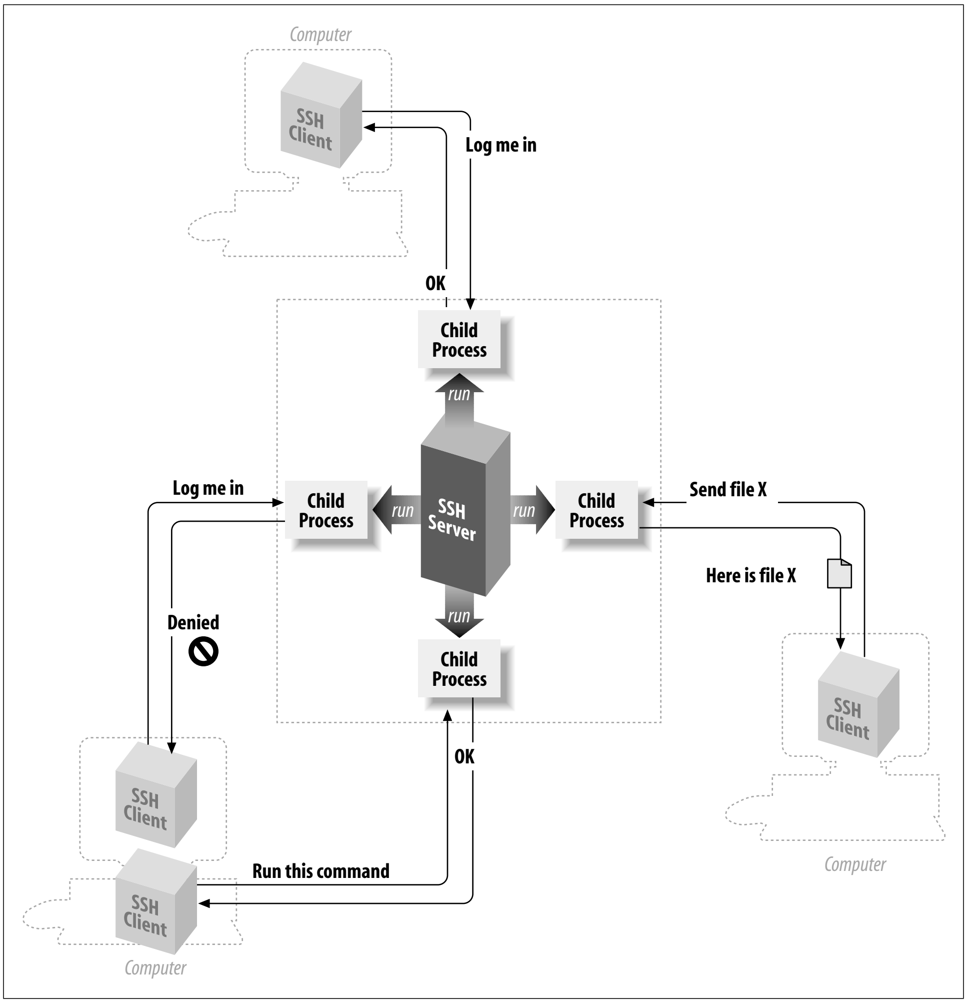


Figure 1-1. SSH architecture

Our description is simplified but should give you a general idea of what SSH does. We'll go into depth later. For now, just remember that SSH clients communicate with SSH servers over encrypted network connections.

SSH software is very common today. It comes with most Linux distributions, Macintosh OS X, Sun Solaris, OpenBSD, and virtually all other Unix-inspired operating systems. Microsoft Windows has plenty of SSH clients and servers, both free and commercial. You can even find it for PalmOS, Commodore Amiga, and most other platforms. [13.3]

Many SSH clients are inspired by old Unix programs called the "r-commands:" *rsh* (remote shell), *rlogin* (remote login), and *rcp* (remote copy). In fact, for many purposes the SSH clients are drop-in replacements for the r-commands, so if you're still using them, switch to SSH immediately! The old r-commands are notoriously insecure, and the SSH learning curve is small.

1.2 What SSH Is Not

Although SSH stands for Secure Shell, it is not a true shell in the sense of the Unix Bourne shell and C shell. It is not a command interpreter, nor does it provide wildcard expansion, command history, and so forth. Rather, SSH creates a channel for running a shell on a remote computer, with end-to-end encryption between the two systems.

SSH is also not a complete security solution—but then, nothing is. It won't protect computers from active break-in attempts or denial-of-service attacks, and it won't eliminate other hazards such as viruses, Trojan horses, and coffee spills. It does, however, provide robust and user-friendly encryption and authentication.

1.3 The SSH Protocol

SSH is a *protocol,* not a product. It is a specification of how to conduct secure communication over a network.*

The SSH protocol covers authentication, encryption, and the integrity of data transmitted over a network, as shown in Figure 1-2. Let's define these terms:

Authentication
: Reliably determines someone's identity. If you try to log into an account on a remote computer, SSH asks for digital proof of your identity. If you pass the test, you may log in; otherwise, SSH rejects the connection.

* Although we say "the SSH protocol," there are actually two incompatible versions of the protocols in common use: SSH-1 (a.k.a. SSH-1.5) and SSH-2. We distinguish these protocols later.

Encryption
Scrambles data so that it is unintelligible except to the intended recipients. This protects your data as it passes over the network.

Integrity
Guarantees the data traveling over the network arrives unaltered. If a third party captures and modifies your data in transit, SSH detects this fact.

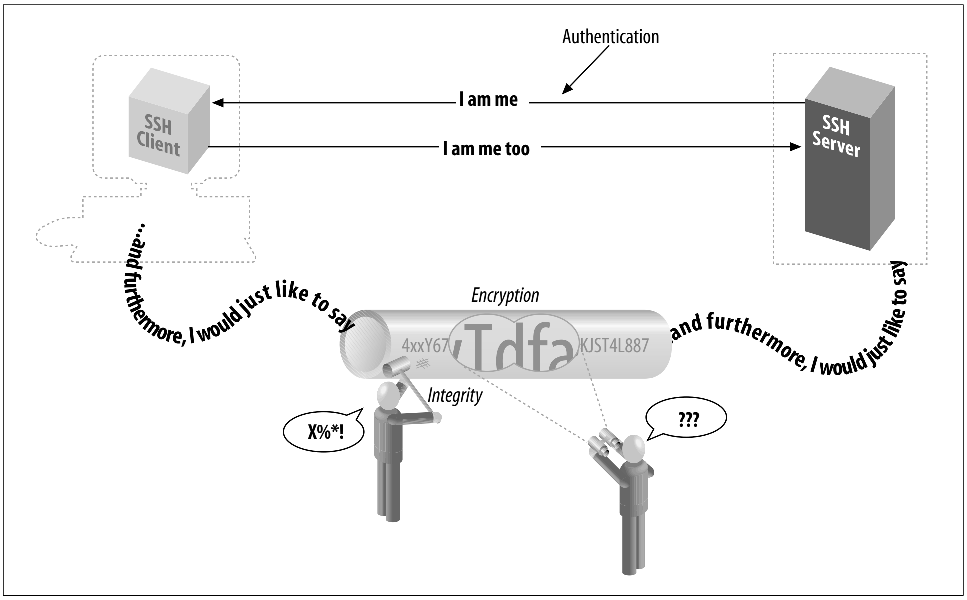


Figure 1-2. Authentication, encryption, and integrity

In short, SSH makes network connections between computers, with strong guarantees that the parties on both ends of the connection are genuine. It also ensures that any data passing over these connections arrives unmodified and unread by eavesdroppers.

1.3.1 Protocols, Products, Clients, and Confusion

The first SSH product, created by Tatu Ylönen for Unix, was simply called "SSH." This caused confusion because SSH was also the name of the protocol. In this book, we use more precise terminology to refer to protocols, products, and programs, summarized in the sidebar "Terminology: SSH Protocols and Products." In short:

- Protocols are denoted with dashes: SSH-1, SSH-2.
- Products are denoted in mixed case, without dashes: OpenSSH, Tectia, PuTTY, etc.
- Client programs are in lowercase: *ssh*, *scp*, *putty*, etc.

Terminology: SSH Protocols and Products

SSH
: A generic term referring to SSH protocols and software products.

SSH-1
: The SSH protocol, Version 1. This is the original protocol, and it has serious limitations, so we do not recommend its use anymore.

SSH-2
: The SSH protocol, Version 2, the most common and secure SSH protocol used today. It is defined by draft standards documents of the IETF SECSH working group. [3.4]

SSH1
: The granddaddy of it all: the original SSH product created by Tatu Ylönen. It implemented (and defined) the SSH-1 protocol and is now obsolete.

SSH2
: The original SSH-2 product, created by Tatu Ylönen and his company, SSH Communications Security (*http://www.ssh.com*).

ssh (all lowercase letters)
: A client program run on the command line and included in many SSH products, for running secure terminal sessions and remote commands. On some systems it might be named *ssh1* or *ssh2*.

OpenSSH
: The product OpenSSH from the OpenBSD project, *http://www.openssh.com*.

Tectia
: The successor to SSH2, this refers to the product suite "SSH Tectia" from SSH Communications Security. We abbreviate the name as simply "Tectia." Since Tectia is available for both Unix and Windows, when we write "Tectia" we generally mean the Unix version unless we say otherwise.

1.4 Overview of SSH Features

So, what can SSH do? Let's run through some examples that demonstrate the major features of SSH, such as secure remote logins, secure file copying, and secure invocation of remote commands.

1.4.1 Secure Remote Logins

Suppose you have login accounts on several computers on the Internet. Common programs like *telnet* let you log into one computer from another, say, from your home PC to your web hosting provider, or from one office computer to another. Unfortunately, *telnet* and similar programs transmit your username and password in plain text over the Internet, where a malicious third party can intercept them.* Additionally, your entire *telnet* session is readable by a network snooper.

Terminology: Networking

Local computer (local host, local machine)
: A computer on which you are logged in and, typically, running an SSH client.

Remote computer (remote host, remote machine)
: A second computer you connect to via your local computer. Typically, the remote computer is running an SSH server and is accessed via an SSH client. As a degenerate case, the local and remote computers can be the same machine.

Local user
: A user logged into a local computer.

Remote user
: A user logged into a remote computer.

Server
: An SSH server program.

Server machine
: A computer running an SSH server program. We sometimes simply write "server" for the server machine when the context makes clear (or irrelevant) the distinction between the running SSH server program and its host machine.

Client
: An SSH client program.

Client machine
: A computer running an SSH client. As with the server terminology, we simply write "client" when the context makes the meaning clear.

~ or $HOME
: A user's home directory on a Unix machine, particularly when used in a file path such as *~/filename*. Most shells recognize ~ as a user's home directory, with the notable exception of the Bourne shell. *$HOME* is recognized by all shells.

SSH completely avoids these problems. Rather than running the insecure *telnet* program, you run the SSH client program *ssh*. To log into an account with the username smith on the remote computer *host.example.com*, use this command:

```
$ ssh -l smith host.example.com
```

The client authenticates you to the remote computer's SSH server using an encrypted connection, meaning that your username and password are encrypted before they leave the local machine. The SSH server then logs you in, and your entire login

* This is true of standard Telnet, but some implementations add security features.

session is encrypted as it travels between client and server. Because the encryption is transparent, you won't notice any differences between *telnet* and the *telnet*-like SSH client.

1.4.2 Secure File Transfer

Suppose you have accounts on two Internet computers, *me@firstaccount.com* and *metoo@secondaccount.com*, and you want to transfer a file from the first to the second account. The file contains trade secrets about your business, however, that must be kept from prying eyes. A traditional file-transfer program, such as *ftp*, doesn't provide a secure solution. A third party can intercept and read the packets as they travel over the network. To get around this problem, you can encrypt the file on *firstaccount.com* with a program such as Pretty Good Privacy (PGP), transfer it via traditional means, and decrypt the file on *secondaccount.com*, but such a process is tedious and nontransparent to the user.

Using SSH, the file can be transferred securely between machines with a single secure copy command. If the file were named *myfile*, the command executed on *firstaccount.com* might be:

```
$ scp myfile metoo@secondaccount.com:
```

When transmitted by *scp*, the file is automatically encrypted as it leaves *firstaccount.com* and decrypted as it arrives on *secondaccount.com*.

1.4.3 Secure Remote Command Execution

Suppose you are a system administrator who needs to run the same command on many computers. You'd like to view the active processes for each user on four different computers—*grape*, *lemon*, *kiwi*, and *melon*—on a local area network using the Unix command */usr/bin/w*. Many SSH clients can run a single remote command if you provide it at the end of the command line. This short shell script does the trick:

```
#!/bin/sh
for machine in grape lemon kiwi melon
do
  ssh $machine /usr/bin/w                    Execute remote command by ssh
done
```

Each *w* command and its results are encrypted as they travel across the network, and strong authentication techniques may be used when connecting to the remote machines.

1.4.4 Keys and Agents

Suppose you have accounts on many computers on a network. For security reasons, you prefer different passwords on all accounts; but remembering so many passwords is difficult. It's also a security problem in itself. The more often you type a password, the more likely you'll mistakenly type it in the wrong place. (Have you ever accidentally typed your password instead of your username, visible to the world? Ouch! And on many systems, such mistakes are recorded in a system log file, revealing your password in plain text.) Wouldn't it be great to identify yourself only once and get secure access to all the accounts without continually typing passwords?

SSH has various authentication mechanisms, and the most secure is based on *keys* rather than passwords. Keys are discussed in great detail in Chapter 6, but for now we define a key as a small blob of bits that uniquely identifies an SSH user. For security, a key is kept encrypted; it may be used only after entering a secret *passphrase* to decrypt it.

Using keys, together with a program called an *authentication agent*, SSH can authenticate you to all your computer accounts securely without requiring you to memorize many passwords or enter them repeatedly. It works like this:

1. In advance (and only once), place special, nonsecure files called *public key files* into your remote computer accounts. These enable your SSH clients (*ssh*, *scp*) to access your remote accounts.
2. On your local machine, invoke the *ssh-agent* program, which runs in the background.
3. Choose the key (or keys) you will need during your login session.
4. Load the keys into the agent with the *ssh-add* program. This requires knowledge of each key's secret passphrase.

At this point, you have an *ssh-agent* program running on your local machine, holding your secret keys in memory. You're now done. You have passwordless access to all your remote accounts that contain your public key files. Say goodbye to the tedium of retyping passwords! The setup lasts until you log out from the local machine or terminate *ssh-agent*.

1.4.5 Access Control

Suppose you want to permit another person to use your computer account, but only for certain purposes. For example, while you're out of town you'd like your secretary to read your email but not to do anything else in your account. With SSH, you can give your secretary access to your account without revealing or changing your password, and with only the ability to run the email program. No system-administrator privileges are required to set up this restricted access. (This topic is the focus of Chapter 8.)

1.4.6 Port Forwarding

SSH can increase the security of other TCP/IP-based applications such as *telnet*, *ftp*, and the X Window System. A technique called *port forwarding* or *tunneling* reroutes a TCP/IP connection to pass through an SSH connection, transparently encrypting it end to end. Port forwarding can also pass such applications through network firewalls that otherwise prevent their use.

Suppose you are logged into a machine away from work and want to access the internal news server at your office, *news.yoyodyne.com*. The Yoyodyne network is connected to the Internet, but a network firewall blocks incoming connections to most ports, particularly port 119, the news port. The firewall does allow incoming SSH connections, however, since the SSH protocol is secure enough that even Yoyodyne's rabidly paranoid system administrators trust it. SSH can establish a secure tunnel on an arbitrary local TCP port—say, port 3002—to the news port on the remote host. The command might look a bit cryptic at this early stage, but here it is:

```
    $ ssh -L 3002:localhost:119 news.yoyodyne.com
```

This says "*ssh*, please establish a secure connection from TCP port 3002 on my local machine to TCP port 119, the news port, on *news.yoyodyne.com*." So, in order to read news securely, configure your news-reading program to connect to port 3002 on your local machine. The secure tunnel created by *ssh* automatically communicates with the news server on *news.yoyodyne.com*, and the news traffic passing through the tunnel is protected by encryption. [9.1]

1.5 History of SSH

SSH1 and the SSH-1 protocol were developed in 1995 by Tatu Ylönen, a researcher at the Helsinki University of Technology in Finland. After his university network was the victim of a password-sniffing attack earlier that year, Ylönen whipped up SSH1 for himself. When beta versions started gaining attention, however, he realized his security product could be put to wider use.

In July 1995, SSH1 was released to the public as free software with source code, permitting people to copy and use the program without cost. By the end of the year, an estimated 20,000 users in 50 countries had adopted SSH1, and Ylönen was fending off 150 email messages per day requesting support. In response, Ylönen founded SSH Communications Security Corp., (SCS, *http://www.ssh.com/*) in December of 1995 to maintain, commercialize, and continue development of SSH. Today he is a board member and technical advisor to the company.

Also in 1995, Ylönen documented the SSH-1 protocol as an Internet Engineering Task Force (IETF) Internet Draft, which essentially described the operation of the SSH1 software after the fact. It was a somewhat ad hoc protocol with a number of problems and limitations discovered as the software grew in popularity. These problems couldn't be fixed without losing backward compatibility, so in 1996, SCS introduced a new, major version of the protocol, SSH 2.0 or SSH-2, that incorporates new algorithms and is incompatible with SSH-1. In response, the IETF formed a working group called Secure Shell (SECSH) to standardize the protocol and guide its development in the public interest. The SECSH working group submitted the first Internet Draft for the SSH-2.0 protocol in February 1997.

In 1998, SCS released the software product SSH Secure Shell (SSH2), based on the superior SSH-2 protocol. However, SSH2 didn't replace SSH1 in the field: it was missing some features of SSH1 and had a more restrictive license, so many users felt little reason to switch, even though SSH-2 is a better and more secure protocol.

This situation changed with the appearance of OpenSSH (*http://www.openssh.com/*), a free implementation of the SSH-2 protocol from the OpenBSD project (*http://www.openbsd.org/*). It was based on the last free release of the original SSH, 1.2.12, but developed rapidly into one of the reigning SSH implementations in the world. Though many people have contributed to it, OpenSSH is largely the work of software developer Markus Friedl. It has been ported successfully to Linux, Solaris, AIX, Mac OS X, and other operating systems, in tight synchronization with the OpenBSD releases.

SCS has continued to improve its SSH products, in some cases beyond what OpenSSH supports. Its product line now carries the name Tectia. And nowadays there are dozens of SSH implementations, both free and commercial, for virtually all platforms. Millions of people use it worldwide to secure their communications.

1.6 Related Technologies

SSH is popular and convenient, but we certainly don't claim it is the ultimate security solution for all networks. Authentication, encryption, and network security originated long before SSH and have been incorporated into many other systems. Let's survey a few representative systems.

1.6.1 rsh Suite (r-Commands)

The Unix programs *rsh*, *rlogin*, and *rcp*—collectively known as the *r-commands*—are the direct ancestors of the SSH clients *ssh*, *slogin*, and *scp*. The user interfaces and visible functionality are nearly identical to their SSH counterparts, except that SSH clients are secure. The r-commands, in contrast, don't encrypt their connections and have a weak, easily subverted authentication model.

An r-command server relies on two mechanisms for security: a network naming service and the notion of "privileged" TCP ports. Upon receiving a connection from a client, the server obtains the network address of the originating host and translates it into a hostname. This hostname must be present in a configuration file on the server, typically */etc/hosts.equiv*, for the server to permit access. The server also checks that the source TCP port number is in the range 1–1023, since these port numbers can be used only by the Unix superuser (or root uid). If the connection passes both checks, the server believes it is talking to a trusted program on a trusted host and logs in the client as whatever user it requests!

These two security checks are easily subverted. The translation of a network address to a hostname is done by a naming service such as Sun's Network Information Service (NIS) or the Internet Domain Name System (DNS). Most implementations and/or deployments of NIS and DNS services have security holes, presenting opportunities to trick the server into trusting a host it shouldn't. Then, a remote user can log into someone else's account on the server simply by having the same username.

Likewise, blind trust in privileged TCP ports represents a serious security risk. A cracker who gains root privilege on a trusted machine can simply run a tailored version of the *rsh* client and log in as any user on the server host. Overall, reliance on these port numbers is no longer trustworthy in a world of desktop computers whose users have administrative access as a matter of course, or whose operating systems don't support multiple users or privileges (such as Windows 9x and Macintosh OS 9).

If user databases on trusted hosts were always synchronized with the server, installation of privileged programs (setuid root) strictly monitored, root privileges guaranteed to be held by trusted people, and the physical network protected, the r-commands would be reasonably secure. These assumptions made sense in the early days of networking, when hosts were few, expensive, and overseen by a small and trusted group of administrators, but they have far outlived their usefulness.

Given SSH's superior security features and that *ssh* is backward-compatible with *rsh* (and *scp* with *rcp*), we see no compelling reason to run the r-commands anymore. Install SSH and be happy.

1.6.2 Pretty Good Privacy (PGP) and GNU Privacy Guard (GnuPG)

PGP is a popular encryption program available for many computing platforms, created by Phil Zimmerman. It can authenticate users and encrypt data files and email messages. GnuPG is a more powerful successor to PGP with less-restrictive licensing.

SSH incorporates some of the same encryption algorithms as PGP and GnuPG, but applied in a different way. PGP is file-based, typically encrypting one file or email message at a time on a single computer. SSH, in contrast, encrypts an ongoing session between networked computers. The difference between PGP and SSH is like that between a batch job and an interactive process.

PGP and SSH are related in another way as well: Tectia can optionally use PGP keys for authentication. [5.4.5]

More PGP and GnuPG information is available at *http://www.pgp.com/* and *http://www.gnupg.org/*, respectively.

1.6.3 Kerberos

Kerberos is a secure authentication system for environments where networks may be monitored, and computers aren't under central control. It was developed as part of Project Athena, a wide-ranging research and development effort at the Massachusetts Institute of Technology (MIT). Kerberos authenticates users by way of *tickets,* small sequences of bytes with limited lifetimes, while user passwords remain secure on a central machine.

Kerberos and SSH solve similar problems but are quite different in scope. SSH is lightweight and easily deployed, designed to work on existing systems with minimal changes. To enable secure access from one machine to another, simply install an SSH client on the first and a server on the second, and start the server. Kerberos, in contrast, requires significant infrastructure to be established before use, such as administrative user accounts, a heavily secured central host, and software for networkwide clock synchronization. In return for this added complexity, Kerberos ensures that users' passwords travel on the network as little as possible and are stored only on the central host. SSH sends passwords across the network (over encrypted connections, of course) on each login and stores keys on each host from which SSH is used. Kerberos also serves other purposes beyond the scope of SSH, including a centralized user account database, access control lists, and a hierarchical model of trust.

Another difference between SSH and Kerberos is the approach to securing client applications. SSH can easily secure most TCP/IP-based programs via a technique called port-forwarding. Kerberos, on the other hand, contains a set of programming libraries for adding authentication and encryption to other applications. Developers can integrate applications with Kerberos by modifying their source code to make calls to the Kerberos libraries. The MIT Kerberos distribution comes with a set of common services that have been "kerberized," including secure versions of *telnet*, *ftp*, and *rsh*.

If the features of both Kerberos and SSH sound good, you're in luck: they've been integrated. [11.4] More information on Kerberos can be found at *http://web.mit.edu/kerberos/www/*.

1.6.4 IPSEC and Virtual Private Networks

Internet Protocol Security (IPSEC) is an Internet standard for network security. Developed by an IETF working group, IPSEC comprises authentication and encryption implemented at the IP level. This is a lower level of the network stack than SSH addresses. It is entirely transparent to end users, who don't need to use a particular program such as SSH to gain security; rather, their existing insecure network traffic is protected automatically by the underlying system. IPSEC can securely connect a single machine to a remote network through an intervening untrusted network (such as the Internet), or it can connect entire networks (this is the idea of the Virtual Private Network, or VPN).

SSH is often quicker and easier to deploy as a solution than IPSEC, since SSH is a simple application program, whereas IPSEC requires additions to the host operating systems on both sides if they don't already come with it, and possibly to network equipment such as routers, depending on the scenario. SSH also provides user authentication, whereas IPSEC deals only with individual hosts. On the other hand, IPSEC is more basic protection and can do things SSH can't. For instance, in Chapter 11 we discuss the difficulties of trying to protect the FTP protocol using SSH. If you need to secure an existing insecure protocol such as FTP, which isn't amenable to treatment with SSH, IPSEC is a way to do it.

IPSEC can provide authentication alone, through a means called the Authentication Header (AH), or both authentication and encryption, using a protocol called Encapsulated Security Payload (ESP). Detailed information on IPSEC can be found at *http://www.ietf.org/html.charters/ipsec-charter.html.*

1.6.5 Secure Remote Password (SRP)

The Secure Remote Password (SRP) protocol, created at Stanford University, is a security protocol very different in scope from SSH. It is specifically an authentication protocol, whereas SSH comprises authentication, encryption, integrity, session management, etc., as an integrated whole. SRP isn't a complete security solution in itself, but rather, a technology that can be a part of a security system.

The design goal of SRP is to improve on the security properties of password-style authentication, while retaining its considerable practical advantages. Using SSH public-key authentication is difficult if you're traveling, especially if you're not carrying your own computer, but instead are using other people's machines. You have to carry your private key on a portable storage device and hope that you can get the key into whatever machine you need to use.

Carrying your encrypted private key with you is also a weakness, because if someone steals it, they can subject it to a dictionary attack in which they try to find your passphrase and recover the key. Then you're back to the age-old problem with passwords: to be useful they must be short and memorable, whereas to be secure, they must be long and random.

SRP provides strong two-party mutual authentication, with the client needing only to remember a short password which need not be so strongly random. With traditional password schemes, the server maintains a sensitive database that must be protected, such as the passwords themselves, or hashed versions of them (as in the Unix */etc/passwd* and */etc/shadow* files). That data must be kept secret, since disclosure allows an attacker to impersonate users or discover their passwords through a dictionary attack. The design of SRP avoids such a database and allows passwords to be less random (and therefore more memorable and useful), since it prevents dictionary attacks. The server still has sensitive data that should be protected, but the consequences of its disclosure are less severe.

SRP is also intentionally designed to avoid using encryption algorithms in its operation. Thus it avoids running afoul of cryptographic export laws, which prohibits certain encryption technologies from being shared with foreign countries.

SRP is an interesting technology we hope gains wider acceptance; it is an excellent candidate for an additional authentication method in SSH. The current SRP implementation includes secure clients and servers for the Telnet and FTP protocols for Unix and Windows. More SRP information can be found at *http://srp.stanford.edu/*.

1.6.6 Secure Socket Layer (SSL) Protocol

The Secure Socket Layer (SSL) protocol is an authentication and encryption technique providing security services to TCP clients by way of a Berkeley sockets-style API. It was initially developed by Netscape Communications Corporation to secure the HTTP protocol between web clients and servers, and that is still its primary use, though nothing about it is specific to HTTP. It is on the IETF standards track as RFC-2246, under the name "TLS" for Transport Layer Security.

An SSL participant proves its identity by a *digital certificate,* a set of cryptographic data. A certificate indicates that a trusted third party has verified the binding between an identity and a given cryptographic key. Web browsers automatically check the certificate provided by a web server when they connect by SSL, ensuring that the server is the one the user intended to contact. Thereafter, transmissions between the browser and the web server are encrypted.

SSL is used most often for web applications, but it can also "tunnel" other protocols. It is secure only if a "trusted third party" exists. Organizations known as *certificate authorities* (CAs) serve this function. If a company wants a certificate from the CA, the company must prove its identity to the CA through other means, such as legal documents. Once the proof is sufficient, the CA issues the certificate.

For more information, visit the OpenSSL project at *http://www.openssl.org/*.

1.6.7 SSL-Enhanced Telnet and FTP

Numerous TCP-based communication programs have been enhanced with SSL, including *telnet* (e.g., SSLtelnet, SRA telnet, SSLTel, STel) and *ftp* (SSLftp), providing some of the functionality of SSH. Though useful, these tools are fairly single-purpose and typically are patched or hacked versions of programs not originally written for secure communication. The major SSH implementations, on the other hand, are more like integrated toolsets with diverse uses, written from the ground up for security.

1.6.8 stunnel

stunnel is an SSL tool created by Micha Trojnara of Poland. It adds SSL protection to existing TCP-based services in a Unix environment, such as POP or IMAP servers, without requiring changes to the server source code. It can be invoked from *inetd* as a wrapper for any number of service daemons or run standalone, accepting network connections itself for a particular service. *stunnel* performs authentication and authorization of incoming connections via SSL; if the connection is allowed, it runs the server and implements an SSL-protected session between the client and server programs.

This is especially useful because certain popular applications have the option of running some client/server protocols over SSL. For instance, email clients like Microsoft Outlook and Mozilla Mail can connect to POP, IMAP, and SMTP servers using SSL. For more *stunnel* information, see *http://www.stunnel.org/*.

1.6.9 Firewalls

A *firewall* is a hardware device or software program that prevents certain data from entering or exiting a network. For example, a firewall placed between a web site and the Internet might permit only HTTP and HTTPS traffic to reach the site. As another example, a firewall can reject all TCP/IP packets unless they originate from a designated set of network addresses.

Firewalls aren't a replacement for SSH or other authentication and encryption approaches, but they do address similar problems. The techniques may be used together.

1.7 Summary

SSH is a powerful, convenient approach to protecting communications on a computer network. Through secure authentication and encryption technologies, SSH supports secure remote logins, secure remote command execution, secure file transfers, access control, TCP/IP port forwarding, and other important features.