SSH, The Secure Shell: The Definitive Guide
By Daniel J. Barrett, Ph.D., Richard Silverman
Publisher: O'Reilly
Pub Date: January 2001
ISBN: 0-596-00011-1
Pages: 558

SSH (Secure Shell) is a popular, robust, TCP/IP-based product for network security and privacy, supporting strong encryption and authentication. This book covers Unix, Windows, and Macintosh implementations of SSH. It shows both system administrators and end users how to install, maintain, and troubleshoot SSH; configure servers and clients in simple and complex ways; apply SSH to practical problems; and protect other TCP applications through forwarding (tunneling).
**Table of Content**

Table of Content................................................................. ii
Preface...................................................................................... vii
Protect Your Network with SSH........................................... vii
Intended Audience............................................................... vii
Reading This Book............................................................... viii
Our Approach........................................................................ ix
Which Chapters Are for You?................................................. ix
Supported Platforms............................................................ x
Disclaimers........................................................................... x
Conventions Used in This Book............................................. xi
Comments and Questions...................................................... xi
Acknowledgments............................................................... xi

Chapter 1. Introduction to SSH........................................... 1
  1.1 What Is SSH?............................................................... 1
  1.2 What SSH Is Not......................................................... 2
  1.3 The SSH Protocol....................................................... 3
  1.4 Overview of SSH Features......................................... 5
  1.5 History of SSH........................................................... 8
  1.6 Related Technologies................................................ 10
  1.7 Summary...................................................................... 14

Chapter 2. Basic Client Use............................................... 15
  2.1 A Running Example.................................................... 15
  2.2 Remote Terminal Sessions with ssh........................... 15
  2.3 Adding Complexity to the Example............................ 17
  2.4 Authentication by Cryptographic Key......................... 20
  2.5 The SSH Agent.......................................................... 25
  2.6 Connecting Without a Password or Passphrase.......... 29
  2.7 Miscellaneous Clients............................................... 30
  2.8 Summary...................................................................... 32

Chapter 3. Inside SSH......................................................... 33
  3.1 Overview of Features.................................................. 33
  3.2 A Cryptography Primer.............................................. 35
  3.3 The Architecture of an SSH System............................ 38
  3.4 Inside SSH-1.............................................................. 40
  3.5 Inside SSH-2.............................................................. 56
  3.6 As-User Access (userfile)............................................ 67
  3.7 Randomness.............................................................. 67
  3.8 SSH and File Transfers (scp and sftp)......................... 69
  3.9 Algorithms Used by SSH........................................... 72
  3.10 Threats SSH Can Counter........................................ 78
  3.11 Threats SSH Doesn't Prevent.................................... 80
  3.12 Summary.................................................................... 83

Chapter 4. Installation and Compile-Time Configuration........ 84
  4.1 SSH1 and SSH2......................................................... 84
  4.2 F-Secure SSH Server................................................ 102
  4.3 OpenSSH................................................................. 103
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.8 ssh-agent Options</td>
<td>424</td>
</tr>
<tr>
<td>2.9 ssh-add Options</td>
<td>424</td>
</tr>
<tr>
<td>2.10 Identity and Authorization Files</td>
<td>424</td>
</tr>
<tr>
<td>2.11 Environment Variables</td>
<td>425</td>
</tr>
<tr>
<td>Colophon</td>
<td>426</td>
</tr>
</tbody>
</table>
Preface

Privacy is a basic human right, but on today's computer networks, privacy isn't guaranteed. Much of the data that travels on the Internet or local networks is transmitted as plain text, and may be captured and viewed by anybody with a little technical know-how. The email you send, the files you transmit between computers, even the passwords you type may be readable by others. Imagine the damage that can be done if an untrusted third party—a competitor, the CIA, your in-laws—intercepted your most sensitive communications in transit.

Network security is big business as companies scramble to protect their information assets behind firewalls, establish virtual private networks (VPNs), and encrypt files and transmissions. But hidden away from all the bustle, there is a small, unassuming, yet robust solution many big companies have missed. It's reliable, reasonably easy to use, cheap, and available for most of today's operating systems.

It's SSH, the Secure Shell.

Protect Your Network with SSH

SSH is a low-cost, software-based solution for keeping prying eyes away from the data on a network. It doesn't solve every privacy and security problem, but it eliminates several of them effectively. Its major features are:

- A secure, client/server protocol for encrypting and transmitting data over a network
- Authentication (recognition) of users by password, host, or public key, plus optional integration with other popular authentication systems, including Kerberos, SecurID, PGP, TIS Gauntlet, and PAM
- The ability to add security to insecure network applications such as Telnet, FTP, and many other TCP/IP-based programs and protocols
- Almost complete transparency to the end user
- Implementations for most operating systems

Intended Audience

We've written this book for system administrators and technically minded users. Some chapters are suitable for a wide audience, while others are thoroughly technical and intended for computer and networking professionals.

End-User Audience

Do you have two or more computer accounts on different machines? SSH lets you connect one to another with a high degree of security. You can copy files between accounts, remotely log into one account from the other, or execute remote commands, all with the confidence that nobody can intercept your username, password, or data in transit.

Do you connect from a personal computer to an Internet service provider (ISP)? In particular, do you connect to a Unix shell account at your ISP? If so, SSH can make this connection significantly
more secure. An increasing number of ISPs are running SSH servers for their users. In case your
ISP doesn't, we'll show you how to run a server yourself.

Do you develop software? Are you creating distributed applications that must communicate over a
network securely? Then don't reinvent the wheel: use SSH to encrypt the connections. It's a solid
technology that may reduce your development time.

Even if you have only a single computer account, as long as it's connected to a network, SSH can
still be useful. For example, if you've ever wanted to let other people use your account, such as
family members or employees, but didn't want to give them unlimited use, SSH can provide a
carefully controlled, limited access channel into your account.

Prerequisites

We assume you are familiar with computers and networking as found in any modern business
office or home system with an Internet connection. Ideally, you are familiar with the Telnet and
FTP applications. If you are a Unix user, you should be familiar with the programs rsh, rlogin, and
rcp, and with the basics of writing shell scripts.

System-Administrator Audience

If you're a Unix system administrator, you probably know that the Berkeley r-commands (rsh, rcp,
rlogin, rexec, etc.) are inherently insecure. SSH provides secure, drop-in replacements,
eliminates .rhosts and hosts.equiv files, and can authenticate users by cryptographic key. SSH also
can increase the security of other TCP/IP-based applications on your system by transparently
"tunneling" them through SSH encrypted connections. You will love SSH.

Prerequisites

In addition to the end-user prerequisites in the previous section, you should be familiar with Unix
accounts and groups, networking concepts such as TCP/IP and packets, and basic encryption
techniques

Reading This Book

This book is roughly divided into three parts. The first three chapters are a general introduction to
SSH, first at a high level for all readers (Chapter 1 and Chapter 2), and then in detail for technical
readers (Chapter 3).

The next nine chapters cover SSH for Unix. The first two (Chapter 4 and Chapter 5) cover SSH
installation and serverwide configuration for system administrators. The next four (Chapter 6-
Chapter 9) cover advanced topics for end users, including key management, client configuration,
per-account server configuration, and forwarding. We complete the Unix sequence with our
recommended setup (Chapter 10), some detailed case studies (Chapter 11), and troubleshooting
tips (Chapter 12).

The remaining chapters cover SSH products for Windows and the Macintosh, plus brief overviews
of implementations for other platforms (Chapter 13).

Each section in the book is numbered, and we provide cross-references throughout the text. If
further details are found in Section 7.1.3.2, we use the notation [Section 7.1.3.2] to indicate it.
Our Approach

This book is organized by concept rather than syntax. We begin with an overview and progressively lead you deeper into the functionality of SSH. So we might introduce a topic in Chapter 1, show its basic use in Chapter 2, and reveal advanced uses in Chapter 7. If you would prefer the whole story at once, Appendix B presents all commands and their options in one location.

We focus strongly on three levels of server configuration, which we call compile-time, serverwide, and per-account configuration. Compile-time configuration (Chapter 4) means selecting appropriate options when you build the SSH clients and servers. Serverwide configuration (Chapter 5) applies when the SSH server is run and is generally done by system administrators, while per-account configuration (Chapter 8) can be done any time by end users. It’s vitally important for system administrators to understand the relationships and differences among these three levels. Otherwise, SSH may seem like a morass of random behaviors.

Although the bulk of material focuses on Unix implementations of SSH, you don't have to be a Unix user to understand it. Fans of Windows and Macintosh may stick to the later chapters devoted to their platforms, but a lot of the meaty details are in the Unix chapters so we recommend reading them, at least for reference.

Which Chapters Are for You?

We propose several "tracks" for readers with different interests and skills:

System administrators

Chapter 3-Chapter 5 and Chapter 10 are the most important for understanding SSH and how to build and configure servers. However, as the administrator of a security product, you should read the whole book.

Unix users (not system administrators)

Chapter 1-Chapter 2 provide an overview, and Chapter 6 through Chapter 9 discuss SSH clients in depth.

Windows end users

Read Chapter 1, Chapter 2, and Chapter 13 through Chapter 16, for starters, and then others as your interests guide you.

Macintosh end users

Read Chapter 1, Chapter 2, Chapter 13, Chapter 16, and Chapter 17, for starters, and then others as your interests guide you.

Users of other computer platforms
Read Chapter 1, Chapter 2, and Chapter 13, for starters, and then others as your interests guide you.

Even if you are experienced with SSH, you will likely find value in Chapter 3-Chapter 12. We cover significant details the Unix manpages leave unclear or unmentioned, including major concepts, compile-time flags, server configuration, and forwarding.

Supported Platforms

This book covers Unix, Windows, and Macintosh implementations of SSH. Products are also available for the Amiga, BeOs, Java, OS/2, Palm Pilot, VMS, and Windows CE, and although we don't cover them, their principles are the same.

This book is current for the following Unix SSH versions:

<table>
<thead>
<tr>
<th>SSH1</th>
<th>1.2.30</th>
</tr>
</thead>
<tbody>
<tr>
<td>F-Secure SSH1</td>
<td>1.3.7</td>
</tr>
<tr>
<td>OpenSSH</td>
<td>2.2.0</td>
</tr>
<tr>
<td>SSH Secure Shell (a.k.a. SSH2)</td>
<td>2.3.0</td>
</tr>
<tr>
<td>F-Secure SSH2</td>
<td>2.0.13</td>
</tr>
</tbody>
</table>

The F-Secure products for Unix differ little from SSH1 and SSH2, so we won't discuss them separately except for unique features. See Appendix B for a summary of the differences.

Version information for non-Unix products is found in their respective chapters.

Disclaimers

We identify some program features as "undocumented." This means the feature isn't mentioned in the official documentation but works in the current release and/or is clear from the program source code. Undocumented features may not be officially supported by the software authors and can disappear in later releases.

Conventions Used in This Book

This book uses the following typographic conventions:

**Constant width**

For configuration files, things that can be found in configuration files (such as keywords and configuration file options), source code, and interactive terminal sessions.

**Constant width italic**

For replaceable parameters on command lines or within configuration files.
For filenames, URLs, hostnames, command names, command-line options, and new terms where they are defined.

$A_K$

In figures, the object labeled A has been secured using a cryptographic key labeled K. "Secured" means encrypted, signed, or some more complex relationship, depending on the context. If A is secured using multiple keys (say K and L), they will be listed in the subscript, separated by commas: $A_{K,L}$.

This icon designates a note, which is an important aside to the nearby text.

This icon designates a warning relating to the nearby text.

Comments and Questions

The information in this book has been tested and verified, but you may find that features have changed (or even find mistakes!). You can send any errors you find, as well as suggestions for future editions, to:

O'Reilly & Associates, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
(800) 998-9938 (in the United States or Canada)
(707) 829-0515 (international/local)
(707) 829-0104 (fax)

There is a web page for this book, which lists errata, examples, or any additional information. You can access this page at:

http://www.oreilly.com/catalog/sshtdg/

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about books, conferences, software, Resource Centers, and the O'Reilly Network, see the O'Reilly web site at:

http://www.oreilly.com/

Acknowledgments
First and foremost, we'd like to thank O'Reilly & Associates for the opportunity to write this book, especially our editor, Mike Loukides, who let us stretch the schedule to cover advanced topics in depth. We thank Frank Willison for believing in our idea, Christien Shangraw for administrative excellence and for heroically performing the first typesetting pass, Mike Sierra for tools and advice, and Rob Romano for turning our hasty sketches into polished illustrations.

We thank our excellent technical review team for their thorough reading and insightful comments: Anne Carasik, Markus Friedl, Joseph Galbraith, Sergey Okhapkin, Jari Ollikka, Niels Provos, Theo de Raadt, Jim Sheafer, Drew Simonis, Mike Smith, and Dug Song.

Big thanks to the vendors and developers of SSH products who provided us with free copies and answered our questions: Tatu Ylönen, Anne Carasik, and Arlinda Sipilä (SSH Communication Security, Ltd.); Sami Sumkin, Heikki Nousiainen, Petri Nyman, Hannu Eloranta, and Alexander Sayer (F-Secure Corporation); Dan Rask (Van Dyke Technologies, Inc.); Gordon Chaffee (Windows SSH port); Ian Goldberg (Top Gun SSH); Douglas Mak (FiSSH); Jonas Walldén (NiftyTelnet SSH); and Stephen Pendleton (sshCE). SSH Communication Security also gave us permission to include the sshregex manpage (Appendix A) and the sshdebug.h error codes (Table 5-6).

We thank Rob Figenbaum, James Mathiesen, and J.D. Paul for tips and inspirations incorporated into the text; and Chuck Bogorad, Ben Gould, David Primmer, and Brandon Zehm for their web pages about SSH on NT. Richard Silverman would like to thank his co-workers at the company formerly known as, especially Michelle Madelien, for being very flexible and accommodating with his erratic hours and behavior while working on this tome. He would also like to thank Deborah Kaplan for her judicious and inspired application of the LART. Lastly, we thank the many contributors to comp.security.ssh on Usenet, for asking good questions that improved the book, especially Chapter 12.
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 one or more PCs 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` and `rcp` for file transfers, `telnet` and `rlogin` 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. 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 it. When the data reaches its intended recipient, SSH 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.

An SSH-based product might include clients, servers, or both. Unix products generally contain both clients and servers; those on other platforms are usually just clients, though Windows-based servers are beginning to appear.

If you're a Unix user, think of SSH as a secure form of the Unix r-commands: *rsh* (remote shell), *rlogin* (remote login), and *rcp* (remote copy). In fact, the original SSH for Unix includes the similarly named commands *ssh*, *scp*, and *slogin* as secure, drop-in replacements for the r-commands. Yes, you can finally get rid of those insecure *.rhosts* and *hosts.equiv* files! (Though SSH can work with them as well, if you like.) If you're still using the r-commands, switch to SSH immediately: the learning curve is small, and security is far better.

### 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, in the manner of the Unix *rsh* command, but with end-to-end encryption between the local and remote computer.
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. [2]

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 will distinguish these protocols later.

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.

Encryption

Scrambles data so 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

SSH-based products—i.e., products that implement the SSH protocol—exist for many flavors of Unix, Windows, Macintosh, and other operating systems. Both freely distributable and commercial products are available. [Section 13.3]

The first SSH product, created by Tatu Ylönen for Unix, was simply called "SSH." This causes confusion because SSH is also the name of the protocol. Some people call Ylönen's software "Unix SSH,” but other Unix-based implementations are now available so the name is unsatisfactory. In this book, we use more precise terminology to refer to protocols, products, and programs, summarized in Sidebar "Terminology: SSH Protocols and Products". In short:

- Protocols are denoted with dashes: SSH-1, SSH-2.
- Products are denoted in uppercase, without dashes: SSH1, SSH2.
- Client programs are in lowercase: ssh, ssh1, ssh2, etc.

### Terminology: SSH Protocols and Products

**SSH**

A generic term referring to SSH protocols or software products.

**SSH-1**

The SSH protocol, Version 1. This protocol went through several revisions, of which 1.3 and 1.5 are the best known, and we will write **SSH-1.3** and **SSH-1.5** should the distinction be necessary.

**SSH-2**

The SSH protocol, Version 2, as defined by several draft standards documents of the IETF SECSH working group. [Section 3.5.1]

**SSH1**

Tatu Ylönen's software implementing the SSH-1 protocol; the original SSH. Now distributed and maintained (minimally) by SSH Communications Security, Inc.

**SSH2**

The "SSH Secure Shell" product from SSH Communications Security, Inc. ([http://www.ssh.com](http://www.ssh.com)). This is a commercial SSH-2 protocol implementation, though it is licensed free of charge in some circumstances.

**ssh (all lowercase letters)**

A client program included in SSH1, SSH2, OpenSSH, F-Secure SSH, and other products, for running secure terminal sessions and remote commands. In SSH1
and SSH2, it is also named ssh1 or ssh2, respectively.

OpenSSH

The product OpenSSH from the OpenBSD project (see http://www.openssh.com/), which implements both the SSH-1 and SSH-2 protocols.

OpenSSH/1

OpenSSH, referring specifically to its behavior when using the SSH-1 protocol.

OpenSSH/2

OpenSSH, referring specifically to its behavior when using the SSH-2 protocol.

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. We use SSH1 in the examples, but all are possible with OpenSSH, SSH2, and F-Secure SSH.

1.4.1 Secure Remote Logins

Suppose you have accounts on several computers on the Internet. Typically, you connect from a home PC to your ISP, and then use a telnet program to log into your accounts on other computers. Unfortunately, telnet transmits your username and password in plaintext over the Internet, where a malicious third party can intercept them. Additionally, your entire telnet session is readable by a network snooper.

[3] This is true of standard Telnet, but some implementations add security features.

**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 contact from your local computer. Typically, the remote computer is running an SSH server and is contacted 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 will 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 will 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 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 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, rcp, or even email, 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/ucb/w. Traditionally, one could use rsh, assuming that the rsh daemon, rshd, is configured properly on the remote computers:

```bash
#!/bin/sh
for machine in grape lemon kiwi melon
do
  rsh $machine /usr/ucb/w
done
```

Although this method works, it's insecure. The results of /usr/ucb/w are transmitted as plaintext across the network; if you consider this information sensitive, the risk might be unacceptable. Worse, the rsh authentication mechanism is extremely insecure and easily subverted. Using the ssh command instead, you have:

```bash
#!/bin/sh
for machine in grape lemon kiwi melon
do
  ssh $machine /usr/ucb/w
done
```

The syntax is nearly identical, and the visible output is identical, but under the hood, the 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 accidently 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 plaintext.) 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 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 password-less 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:

```bash
$ 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. ([Section 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 that 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 receiving over 150 email messages per day requesting support. In response, Ylönen founded SSH Communications Security, Ltd., (SCS, http://www.ssh.com/) in December of 1995 to maintain, commercialize, and continue development of SSH. Today he is chairman and chief technology officer of 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 SECSH (Secure Shell) 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, for two reasons. First, SSH2 was missing a number of useful, practical features and configuration options of SSH1. Second, SSH2 had a more restrictive license. The original SSH1 had been freely available from Ylönen and the Helsinki University of Technology. Newer versions of SSH1 from SCS were still freely available for most uses, even in commercial settings, as long as the software was not directly sold for profit or offered as a service to customers. SSH2, on the other hand, was a commercial product, allowing gratis use only for qualifying educational and non-profit entities. As a result, when SSH2 first appeared, most existing SSH1 users saw few advantages to SSH2 and continued to use SSH1. As of this writing, three years after the introduction of the SSH-2 protocol, SSH-1 is still the most widely deployed version on the Internet, even though SSH-2 is a better and more secure protocol.

This situation promises to change, however, as a result of two developments: a loosening of the SSH2 license and the appearance of free SSH-2 implementations. As this book went to press in late 2000, SCS broadened the SSH2 license to permit free use by individual contractors working for qualifying noncommercial entities. It also extends free use to the Linux, NetBSD, FreeBSD, and OpenBSD operating systems, in any context at all including a commercial one. At the same time, OpenSSH (http://www.openssh.com/) is gaining prominence as an SSH implementation, developed under the auspices of the OpenBSD project (http://www.openbsd.org/) and freely available under the OpenBSD license. Based on the last free release of the original SSH, 1.2.12, OpenSSH has developed rapidly. Though many people have contributed to it, OpenSSH is largely the work of software developer Markus Friedl. It supports both SSH-1 and SSH-2 in a single set of programs, whereas SSH1 and SSH2 have separate executables, and the SSH-1 compatibility features in SSH2 require both products to be installed. While OpenSSH was developed under OpenBSD, it has been ported successfully to Linux, Solaris, AIX, and other operating systems, in tight synchronization with the main releases. Although OpenSSH is relatively new and missing some features present in SSH1 and SSH2, it is developing rapidly and promises to be a major SSH flavor in the near future.

At press time, development of SSH1 has ceased except for important bug fixes, while development of SSH2 and OpenSSH remains active. Other SSH implementations abound, notably the commercial versions of SSH1 and SSH2 maintained and sold by F-Secure Corporation, and numerous ports and original products for the PC, Macintosh, Palm Pilot, and other operating systems. [Section 13.3] It is estimated there are over two million SSH users worldwide, including hundreds of thousands of registered users of SCS products.
Sometimes we use the term "SSH1/SSH2 and their derivatives." This refers to SCS's SSH1 and SSH2, F-Secure SSH Server (Versions 1 and 2), OpenSSH, and any other ports of the SSH1 or SSH2 code base for Unix or other operating systems. The term doesn't encompass other SSH products (SecureCRT, NiftyTelnet SSH, F-Secure's Windows and Macintosh clients, etc.).

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 SSH1 clients `ssh`, `slogin`, and `scp`. The user interfaces and visible functionality are nearly identical to their SSH1 counterparts, except that SSH1 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 the Macintosh).

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` any more. Install SSH and be happy.
1.6.2 Pretty Good Privacy (PGP)

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.

SSH incorporates some of the same encryption algorithms as PGP, 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.

More PGP information is available at http://www.pgpi.com/.

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 network-wide 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 be easily integrated with programs that use rsh in the background, such as Pine, the popular mail reader. Configure it to use ssh instead of rsh, and the program's remote connections are transparently secure. For programs that open direct network connections, SSH's port-forwarding feature provides another convenient form of integration. 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 Kerberos and SSH both sound good, you're in luck: they've been integrated. More information on Kerberos can be found at:

http://web.mit.edu/kerberos/www/
http://nii.isi.edu/info/kerberos/

[4] SSH2 has moved toward this model as well, organized as a set of libraries implementing the SSH2 protocol and accessed via an API.
1.6.4 IPSEC

Internet Protocol Security (IPSEC) is an evolving 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 Section 11.2, we discuss in detail 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/ids.by.wg/ipsec.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 with you on a diskette and hope that you can get the key into whatever machine you need to use. Oops, you've been given an X terminal. Oh well.

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, both Netscape Communicator and Microsoft Internet Explorer allow you to connect POP, IMAP, and SMTP servers using SSL. For more stunnel information, see:

http://www.stanton.dtcc.edu/stanton/cs/admin/notes/ssl

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.
Chapter 2. Basic Client Use

SSH is a simple idea, but it has many complex parts. This chapter is designed to get you started with SSH quickly. We cover the basics of SSH's most immediately useful features:

- Logging into a remote computer over a secure connection
- Transferring files between computers over a secure connection

We also introduce authentication with cryptographic keys, a more secure alternative to ordinary passwords. Advanced uses of client programs, such as multiple keys, client configuration files, and TCP port forwarding, will be covered in later chapters.

We use SSH1 and SSH2 (and occasionally OpenSSH) for all examples. If the syntax differs among the products, we'll discuss each of them.

2.1 A Running Example

Suppose you're out of town on a business trip and want to read your email, which sits on a Unix machine belonging to your ISP, shell.isp.com. A friend at a nearby university agrees to let you log into her Unix account on the machine local.university.edu, and then remotely log into yours. For the remote login you could use the telnet or rlogin programs, but as we've seen, this connection between the machines is insecure. (No doubt some subversive college student would grab your password and turn your account into a renegade web server for pirated software and Ani DiFranco MP3s.) Fortunately, both your friend's Unix machine and your ISP's have an SSH product installed.

In the example running through the chapter, we represent the shell prompt of the local machine, local.university.edu, as a dollar sign ($) and the prompt on shell.isp.com as shell.isp.com>.

2.2 Remote Terminal Sessions with ssh

Suppose your remote username on shell.isp.com is "pat". To connect to your remote account from your friend's account on local.university.edu, you type:

```
$ ssh -l pat shell.isp.com
pat's password: ******
Last login: Mon May 24 19:32:51 1999 from quondam.nefertiti.org
You have new mail.
shell.isp.com>
```

This leads to the situation shown in Figure 2-1. The ssh command runs a client that contacts the SSH server on shell.isp.com over the Internet, asking to be logged into the remote account with username pat.[1] You can also provide user@host syntax instead of the -l option to accomplish the same thing:

```
[1] If the local and remote usernames are identical, you can omit the -l option (-l pat) and just type ssh shell.isp.com.
```

[1] If the local and remote usernames are identical, you can omit the -l option (-l pat) and just type ssh shell.isp.com.
On first contact, SSH establishes a secure channel between the client and the server so all transmissions between them are encrypted. The client then prompts for your password, which it supplies to the server over the secure channel. The server authenticates you by checking that the password is correct and permits the login. All subsequent client/server exchanges are protected by that secure channel, including the contents of the email you proceed to read using a mail program on `shell.isp.com`.

It's important to remember that the secure channel exists only between the SSH client and server machines. After logging into `shell.isp.com` via `ssh`, if you then `telnet` or `ftp` to a third machine, `insecure.isp.com`, the connection between `shell.isp.com` and `insecure.isp.com` is not secure. However, you can run another `ssh` client from `shell.isp.com` to `insecure.isp.com`, creating another secure channel, which keeps the chain of connections secure.

We've covered only the simplest use of `ssh`. Chapter 7 goes into far greater depth about its many features and options.

### 2.2.1 File Transfer with `scp`

Continuing the story, suppose that while reading your email, you encounter a message with an attached file you'd like to print. In order to send the file to a local printer at the university, you must first transfer the file to `local.university.edu`. Once again, you reject as insecure the traditional file-transfer programs, such as `ftp` and `rcp`. Instead, you use another SSH client program, `scp`, to copy the file across the network via a secure channel.

First, you write the attachment to a file in your home directory on `shell.isp.com` using your mail client, naming the file `print-me`. When you've finished reading your other email messages, log out of `shell.isp.com`, ending the SSH session and returning to the shell prompt on `local.university.edu`. You're now ready to copy the file securely.

The `scp` program has syntax much like the traditional Unix `cp` program and nearly identical to the insecure `rcp` program. It is roughly:

```
scp name-of-source name-of-destination
```

In this example, `scp` copies the file `print-me` on `shell.isp.com` over the network to a local file in your friend's account on `local.university.edu`, also called `print-me`:

```
$ scp pat@shell.isp.com:print-me print-me
```

The file is transferred over an SSH-secured connection. The source and destination files may be specified not only by filename, but also by username ("pat" in our example) and hostname (`shell.isp.com`), indicating the location of the file on the network. Depending on your needs, various parts of the source or destination name can be omitted, and defaults values used. For
example, omitting the username and the "at" sign (pat@) makes \texttt{scp} assume that the remote username is the same as the local one.

Like \texttt{ssh}, \texttt{scp} prompts for your remote password and passes it to the SSH server for verification. If successful, \texttt{scp} logs into the pat account on \texttt{shell.isp.com}, copies your remote file \texttt{print-me} to the local file \texttt{print-me}, and logs out of \texttt{shell.isp.com}. The local file \texttt{print-me} may now be sent to a printer.

The destination filename need not be the same as the remote one. For example, if you're feeling French, you could call the local file \texttt{imprime-moi}:

\begin{verbatim}
$ scp pat@shell.isp.com:print-me imprime-moi
\end{verbatim}

The full syntax of \texttt{scp} can represent local and remote files in powerful ways, and the program also has numerous command-line options. \cite{Section 7.5}

\section*{2.3 Adding Complexity to the Example}

The preceding example session provided a quick introduction to the most often-used client programs—\texttt{ssh} and \texttt{scp}—in a format to follow while sitting at your computer. Now that you have the basics, let's continue the example but include situations and complications glossed over the first time. These include the "known hosts" security feature and the SSH escape character.

\begin{tcolorbox}
If you're following at the computer as you read, your SSH clients might behave unexpectedly or differently from ours. As you will see throughout the book, SSH implementations are highly customizable, by both yourself and the system administrator, on either side of the secure connection. Although this chapter describes common behaviors of SSH programs based on their installation defaults, your system might be set up differently.

If commands don't work as you expect, try adding the \texttt{-v} ("verbose") command-line option, for example:

\begin{verbatim}
$ ssh -v shell.isp.com
\end{verbatim}

This causes the client to print lots of information about its progress, often revealing the source of the discrepancy.
\end{tcolorbox}

\subsection*{2.3.1 Known Hosts}

The first time an SSH client encounters a new remote machine, it does some extra work and prints a message like the following:

\begin{verbatim}
$ ssh -l pat shell.isp.com
Host key not found from the list of known hosts. Are you sure you want to continue connecting (yes/no)?
\end{verbatim}

Assuming you respond \texttt{yes} (the most common response), the client continues:

\begin{verbatim}
Host 'shell.isp.com' added to the list of known hosts.
\end{verbatim}
This message appears only the first time you contact a particular remote host. The message is a security feature related to SSH's concept of known hosts.

Suppose an adversary wants to obtain your password. He knows you are using SSH, and so he can't monitor your connection by eavesdropping on the network. Instead, he subverts the naming service used by your local host so that the name of your intended remote host, shell.isp.com, translates falsely to the IP address of a computer run by him! He then installs an altered SSH server on the phony remote host and waits. When you log in via your trusty SSH client, the altered SSH server records your password for the adversary's later use (or misuse, more likely). The bogus server can then disconnect with a preplanned error message such as "System down for maintenance—please try again after 4:00 p.m." Even worse, it can fool you completely by using your password to log into the real shell.isp.com and transparently pass information back and forth between you and the server, monitoring your entire session. This hostile strategy is called a man-in-the-middle attack. [Section 3.10.4] Unless you think to check the originating IP address of your session on the server, you might never notice the deception.

The SSH known-host mechanism prevents such attacks. When an SSH client and server make a connection, each of them proves its identity to the other. Yes, not only does the server authenticate the client, as we saw earlier when the server checked pat's password, but the client also authenticates the server by public-key cryptography. [Section 3.4.1] In short, each SSH server has a secret, unique ID, called a host key, to identify itself to clients. The first time you connect to a remote host, a public counterpart of the host key gets copied and stored in your local account (assuming you responded "yes" to the client's prompt about host keys, earlier). Each time you reconnect to that remote host, the SSH client checks the remote host's identity using this public key.

Of course, it's better to have recorded the server's public host key before connecting to it the first time, since otherwise you are technically open to a man-in-the-middle attack that first time. Administrators can maintain system-wide known-hosts lists for given sets of hosts, but this doesn't do much good for connecting to random new hosts around the world. Until a reliable, widely deployed method of retrieving such keys securely exists (such as secure DNS, or X.509-based public-key infrastructure), this record-on-first-use mechanism is an acceptable compromise.

If authentication of the server fails, various things may happen depending on the reason for failure and the SSH configuration. Typically a warning appears on the screen, ranging from a repeat of the known-hosts message:

Host key not found from the list of known hosts.
Are you sure you want to continue connecting (yes/no)?

to more dire words:

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ WARNING: HOST IDENTIFICATION HAS CHANGED! @@ IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY! Someone could be eavesdropping on you right now (man-in-the-middle attack)! It is also possible that the host key has just been changed. Please contact your system administrator. Add correct host key in <path>/known_hosts to get rid of this message. Agent forwarding is disabled to avoid attacks by corrupted servers. X11 forwarding is disabled to avoid attacks by corrupted servers. Are you sure you want to continue connecting (yes/no)
If you answer yes, ssh allows the connection, but disables various features as a security precaution and doesn't update your personal known-hosts database with the new key; you must do that yourself to make this message go away.

As the text of the message says, if you see this warning, you aren't necessarily being hacked: for example, the remote host may have legitimately changed its host key for some reason. In some cases, even after reading this book, you won't know the cause of these messages. Contact your system administrator if you need assistance, rather than take a chance and possibly compromise your password. We'll cover these issues further when we discuss personal known hosts databases and how to alter the behavior of SSH clients with respect to host keys. [Section 7.4.3]

2.3.2 The Escape Character

Let us return to the shell.isp.com example, just after you'd discovered the attachment in your remote email message and saved it to the remote file print-me. In our original example, you then logged out of shell.isp.com and ran scp to transfer the file. But what if you don't want to log out? If you're using a workstation running a window system, you can open a new window and run scp. But if you're using a lowly text terminal, or you're not familiar with the window system running on your friend's computer, there is an alternative. You can temporarily interrupt the SSH connection, transfer the file (and run any other local commands you desire), and then resume the connection.

ssh supports an escape character, a designated character that gets the attention of the SSH client. Normally, ssh sends every character you type to the server, but the escape character is caught by the client, alerting it that special commands may follow. By default, the escape character is the tilde (~), but you can change it. To reduce the chances of sending the escape character unintentionally, that character must be the first character on the command line, i.e., following a newline (Control-J) or return (Control-M) character. If not, the client treats it literally, not as an escape character.

After the escape character gets the client's attention, the next character entered determines the effect of the escape. For example, the escape character followed by a Control-Z suspends ssh like any other shell job, returning control to the local shell. Such a pair of characters is called an escape sequence. Table 2-1 summarizes the supported escape sequences. It's followed by a list that describes each sequence's meaning.

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Example with &lt;ESC&gt; = ~</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;ESC&gt; ^Z</td>
<td>~ ^Z</td>
<td>Suspend the connection (^Z means Control-Z)</td>
</tr>
<tr>
<td>&lt;ESC&gt; .</td>
<td>~ .</td>
<td>Terminate the connection</td>
</tr>
<tr>
<td>&lt;ESC&gt; #</td>
<td>~ #</td>
<td>List all forwarded connections [2]</td>
</tr>
<tr>
<td>&lt;ESC&gt; &amp;</td>
<td>~ &amp;</td>
<td>Send ssh into the background (when waiting for connections to terminate)</td>
</tr>
<tr>
<td>&lt;ESC&gt; r</td>
<td>~ r</td>
<td>Request rekeying immediately (SSH2 only)</td>
</tr>
<tr>
<td>&lt;ESC&gt;&lt;ESC&gt;</td>
<td>~ ~</td>
<td>Send the escape character (by typing it twice)</td>
</tr>
<tr>
<td>&lt;ESC&gt; ?</td>
<td>~ ?</td>
<td>Print a help message</td>
</tr>
<tr>
<td>&lt;ESC&gt; -</td>
<td>~ -</td>
<td>Disable the escape character (SSH2 only)</td>
</tr>
<tr>
<td>&lt;ESC&gt; V</td>
<td>~ V</td>
<td>Print version information (SSH2 only)</td>
</tr>
<tr>
<td>&lt;ESC&gt; s</td>
<td>~ s</td>
<td>Print statistics about this session (SSH2 only)</td>
</tr>
</tbody>
</table>

[2] For SSH2, this option is documented but not implemented as of Version 2.3.0.
"Suspend the connection" puts ssh into the background, suspended, returning control of the terminal to the local shell. To return to ssh, use the appropriate job control command of your shell, typically \texttt{fg}. While suspended, ssh doesn't run, and if left suspended long enough, the connection may terminate since the client isn't responding to the server. Also, any forwarded connections are similarly blocked while ssh is suspended. [Section 9.2.9]

"Terminate the connection" ends the SSH session immediately. This is most useful if you have lost control of the session: for instance, if a shell command on the remote host has hung and become unkillable. Any X or TCP port forwardings are terminated immediately as well. [Section 9.2.9]

"List all forwarded connections" prints a list of each X forwarding or TCP port forwarding connection currently established. This lists only active instances of forwarding; if forwarding services are available but not currently in use, nothing is listed here.

"Send ssh into the background," like the "suspend connection" command, reconnects your terminal to the shell that started ssh, but it doesn't suspend the ssh process. Instead, ssh continues to run. This isn't ordinarily useful, since the backgrounded ssh process immediately encounters an error.\footnote{The error occurs as ssh attempts to read input from the now disconnected pseudo-terminal.} This escape sequence becomes useful if your ssh session has active, forwarded connections when you log out. Normally in this situation, the client prints a message:

\begin{quote}
Waiting for forwarded connections to terminate...
The following connections are open:
X11 connection from shell.isp.com port 1996
\end{quote}
as it waits for the forwarded connections to close before it exits. While the client is in this state, this escape sequence returns you to the local shell prompt.

"Request rekeying immediately" causes the SSH2 client and server to generate and use some new internal keys for encryption and integrity.

"Send the escape character" tells the client to send a real tilde (or whatever the escape character is) to the SSH server as plaintext, not to interpret it as an escape. "Disable the escape character" prevents further escape sequences from having any effect. The rest of the escape sequences are self-explanatory.

To change the ssh escape character, use the -e command-line option. For example, type the following to make the percent sign (\%) the escape character when connecting to shell.isp.com as user pat:

```
$ ssh -e "\%" -l pat shell.isp.com
```

2.4 Authentication by Cryptographic Key

In our running example, the user pat is authenticated by the SSH server via login password. Passwords, however, have serious drawbacks:

- In order for a password to be secure, it should be long and random, but such passwords are hard to memorize.
- A password sent across the network, even protected by an SSH secure channel, can be captured when it arrives on the remote host if that host has been compromised.
- Most operating systems support only a single password per account. For shared accounts (e.g., a superuser account), this presents difficulties:
Password changes are inconvenient because the new password must be communicated to all people with access to the account.

Tracking usage of the account becomes difficult because the operating system doesn't distinguish between the different users of the account.

To address these problems, SSH supports public-key authentication: instead of relying on the password scheme of the host operating system, SSH may use cryptographic keys. [Section 3.2.2] Keys are more secure than passwords in general and address all the weaknesses mentioned earlier.

### 2.4.1 A Brief Introduction to Keys

A key is a digital identity. It's a unique string of binary data that means, "This is me, honestly, I swear." And with a little cryptographic magic, your SSH client can prove to a server that its key is genuine, and you are really you.

An SSH identity uses a pair of keys, one private and one public. The private key is a closely guarded secret only you have. Your SSH clients use it to prove your identity to servers. The public key is, like the name says, public. You place it freely into your accounts on SSH server machines. During authentication, the SSH client and server have a little conversation about your private and public key. If they match (according to a cryptographic test), your identity is proven, and authentication succeeds.

The following sequence demonstrates the conversation between client and server. [Section 3.4.1] (It occurs behind the scenes, so you don't need to memorize it or anything; we just thought you might be interested.)

1. Your client says, "Hey server, I'd like to connect by SSH to an account on your system, specifically, the account owned by user smith."
2. The server says, "Well, maybe. First, I challenge you to prove your identity!" And the server sends some data, known as a challenge, to the client.
3. Your client says, "I accept your challenge. Here is proof of my identity. I made it myself by mathematically using your challenge and my private key." This response to the server is called an authenticator.
4. The server says, "Thanks for the authenticator. I will now examine the smith account to see if you may enter." Specifically, the server checks smith's public keys to see if the authenticator "matches" any of them. (The "match" is another cryptographic operation.) If so, the server says, "OK, come on in!" Otherwise, the authentication fails.

Before you can use public-key authentication, some setup is required:

1. You need a private key and a public key, known collectively as a key pair. You also need a secret passphrase to protect your private key. [Section 2.4.2]
2. You need to install your public key on an SSH server machine. [Section 2.4.3]

### 2.4.2 Generating Key Pairs with ssh-keygen

To use cryptographic authentication, you must first generate a key pair for yourself, consisting of a private key (your digital identity that sits on the client machine) and a public key (that sits on the server machine). To do this, use the ssh-keygen program. Its behavior differs for SSH1, SSH2, and OpenSSH. On an SSH1 system, the program is called ssh-keygen or ssh-keygen1. When you invoke it, ssh-keygen creates an RSA key pair and asks you for a secret passphrase to protect the private key. [4]

[4] RSA is an encryption algorithm for SSH keys, among other things. [Section 3.9.1] DSA is another, as you'll see later.
$ ssh-keygen
Initializing random number generator...
Generating p: ..................................++ (distance 1368)
Generating q: ....++ (distance 58)
Computing the keys...
Testing the keys...
Key generation complete.
Enter file in which to save the key (/home/pat/.ssh/identity):
Enter passphrase: **************
Enter the same passphrase again: **************
Your identification has been saved in identity.
Your public key is:
1024 35 1127272195779936880509167858732970485872567486703821636830
19500993487602321886571857276011133767701853088352661186539160906
9214986989240214507621864063548908730298546478215446737245984456708
963106607710761174114663544313782992987840457273825436579285836220
2493395730648451296601594344979290457421809236729 path@shell.isp.com
Your public key has been saved in identity.pub.

On SSH2 systems, the command is either ssh-keygen or ssh-keygen2, and its behavior is a bit different and produces either a DSA key (the default) or an RSA key:

$ ssh-keygen2
Generating 1024-bit dsa key pair
  1 ..oOo.oOo.oO  
  2 o.oOo.oOo.oO  
  3 o.oOo.oOo.oO  
  4 o.oOo.oOo.oO  
Key generated.
1024-bit dsa, created by pat@shell.isp.com Mon Mar 20 13:01:15 2000
Passphrase : **************
Again : **************
Private key saved to /home/pat/.ssh2/id_dsa_1024_a
Public key saved to /home/pat/.ssh2/id_dsa_1024_a.pub

The OpenSSH version of ssh-keygen also can produce either RSA or DSA keys, defaulting to RSA. Its operation is similar to that of ssh-keygen1.

Normally, ssh-keygen performs all necessary mathematics to generate a key, but on some operating systems you might be asked to assist it. Key generation requires some random numbers, and if your operating system doesn't supply a random-number generator, you may be asked to type some random text. ssh-keygen uses the timings of your keystrokes to initialize its internal random-number generator. On a 300-MHz Pentium system running Linux, generating a 1024-bit RSA key takes about three seconds; if your hardware is slower than this or heavily loaded, generation may take significantly longer, up to a minute or more. It can also take longer if the process runs out of random bits, and ssh-keygen has to wait to collect more.

ssh-keygen then creates your local SSH directory (~/.ssh for SSH1 and OpenSSH or ~/.ssh2 for SSH2) if it doesn't already exist, and stores the private and public components of the generated key in two files there. By default, their names are identity and identity.pub (SSH1, OpenSSH) or id_dsa_1024_a and id_dsa_1024_a.pub (SSH2). SSH clients consider these to be your default identity for authentication purposes.

Never reveal your private key and passphrase to anyone else. They are just as sensitive as your login password. Anyone possessing them can log in as you!
When created, the identity file is readable only by your account, and its contents are further protected by encrypting them with the passphrase you supplied during generation. We say "passphrase" instead of "password" both to differentiate it from a login password, and to stress that spaces and punctuation are allowed and encouraged. We recommend a passphrase at least 10 - 15 characters long and not a grammatical sentence.

`ssh-keygen` has numerous options for managing keys: changing the passphrase, choosing a different name for the key file, and so forth. [Section 6.2]

### 2.4.3 Installing a Public Key on an SSH Server Machine

When passwords are used for authentication, the host operating system maintains the association between the username and the password. For cryptographic keys, you must set up a similar association manually. After creating the key pair on the local host, you must install your public key in your account on the remote host. A remote account may have many public keys installed for accessing it in various ways.

Returning to our running example, you must install a public key into the "pat" account on `shell.isp.com`. This is done by editing a file in the SSH configuration directory:


[^5]: OpenSSH uses `authorized_keys2` for SSH-2 connections. For simplicity, we'll discuss OpenSSH later. [Section 8.2.3]

For SSH1 or OpenSSH, create or edit the file `~/.ssh/authorized_keys` and append your public key, i.e., the contents of the `identity.pub` file you generated on the local machine. A typical `authorized_keys` file contains a list of public key data, one key per line. The example contains only two public keys, each on its own line of the file, but they are too long to fit on this page. The line breaks inside the long numbers are printing artifact; if they were actually in the file, it would be incorrectly formatted and wouldn't work:

```
1024 35 869751124798752578486656226224505474204292260357215616159982327
587956883143362147028876494426516682677550219425827002174890309672203
21970093718777797970586410754910660881120414204660066790196940691100
768682518506600601481676686828742807110888494083109892341424756942985
20575977312478025518391 my personal key
1024 37 1140868200916227508775331982659387253607752793422843620910258
618820621996941824516069919525136671585267698112659690736259150374130
846896838697083490981532877352706061107257845462743793679411866715467
672826112629198483222167783914580965674001731023827042965273839192998
250061795483568436433123392629 my work key
```

These are RSA public keys: the first number in each entry is the number of bits in the key, while the second and third are RSA-specific parameters called the public exponent and modulus. After these comes an arbitrary amount of text treated as a comment. [Section 8.2.1]

For SSH2, you need to edit two files, one on the client machine and one on the server machine. On the client machine, create or edit the file `~/.ssh2/identification` and insert a line to identify your private key file:

```
IdKey id_dsa_1024_a
```

On the server machine, create or edit the file `~/.ssh2/authorization`, which contains information about public keys, one per line. But unlike SSH1's `authorized_keys` file, which contains copies of the public keys, the `authorization` file lists only the filename of the key:
Key id_dsa_1024_a.pub

Finally, copy id_dsa_1024_a.pub from your local machine to the remote SSH2 server machine, placing it in ~/.ssh2.

Regardless of which SSH implementation you use, make sure your remote SSH directory and associated files are writable only by your account.[6]

We make files world-readable and directories world-searchable, to avoid NFS problems. [Section 10.7.2]

# SSH1, OpenSSH
$ chmod 755 ~/.ssh
$ chmod 644 ~/.ssh/authorized_keys

# OpenSSH only
$ chmod 644 ~/.ssh/authorized_keys2

# SSH2 only
$ chmod 755 ~/.ssh2
$ chmod 644 ~/.ssh2/id_dsa_1024_a.pub
$ chmod 644 ~/.ssh2/authorization

The SSH server is picky about file and directory permissions and may refuse authentication if the remote account’s SSH configuration files have insecure permissions. [Section 5.4.2.1]

You are now ready to use your new key to access the "pat" account:

# SSH1, SSH2, OpenSSH; output shown is for SSH1
$ ssh -l pat shell.isp.com
Enter passphrase for RSA key 'Your Name <you@local.org>': ************
Last login: Mon May 24 19:44:21 1999 from quincunx.nefertiti.org
You have new mail.
shell.isp.com>

If all goes well, you are logged into the remote account. Figure 2-2 shows the entire process.

Figure 2.2. Public-key authentication

Note the similarity to the earlier example with password authentication. [Section 2.2] On the surface, the only difference is that you provide the passphrase to your private key, instead of providing your login password. Underneath, however, something quite different is happening. In password authentication, the password is transmitted to the remote host. With cryptographic
authentication, the passphrase serves only to decrypt the private key to create an authenticator. [Section 2.4.1]

Public-key authentication is more secure than password authentication because:

- It requires two secret components—the identity file on disk, and the passphrase in your head—so both must be captured in order for an adversary to access your account. Password authentication requires only one component, the password, which might be easier to steal.
- Neither the passphrase nor the key is sent to the remote host, just the authenticator discussed earlier. Therefore, no secret information is transmitted off the client machine.
- Machine-generated cryptographic keys are infeasible to guess. Human-generated passwords are routinely cracked by a password-guessing technique called a dictionary attack. A dictionary attack may be mounted on the passphrase as well, but this requires stealing the private key file first.

A host's security can be greatly increased by disabling password authentication altogether and permitting only SSH connections by key.

### 2.4.4 If You Change Your Key

Suppose you have generated a key pair, *identity* and *identity.pub*, and copied *identity.pub* to a bunch of SSH server machines. All is well. Then one day, you decide to change your identity, so you run `ssh-keygen` a second time, overwriting *identity* and *identity.pub*. Guess what? Your previous public key file is now invalid, and you must copy the new public key to all those SSH server machines again. This is a maintenance headache, so think carefully before changing (destroying!) a key pair. Some caveats:

- You are not limited to one key pair. You can generate as many as you like, stored in different files, and use them for diverse purposes. [Section 6.4]
- If you just want to change your passphrase, you don't have to generate a new key pair. `ssh-keygen` has command-line options for replacing the passphrase of an existing key: `-p` for SSH1 and OpenSSH [Section 6.2.1] and `-e` for SSH2 [Section 6.2.2]. In this case your public key remains valid since the private key hasn't changed, just the passphrase for decrypting it.

### 2.5 The SSH Agent

Each time you run `ssh` or `scp` with public-key authentication, you have to retype your passphrase. The first few times you might not mind, but eventually this retyping gets annoying. Wouldn't it be nicer to identify yourself just once and have `ssh` and `scp` remember your identity until further notice (for example, until you log out), not prompting for your passphrase? In fact, this is just what an *SSH agent* does for you.

An agent is a program that keeps private keys in memory and provides authentication services to SSH clients. If you preload an agent with private keys at the beginning of a login session, your SSH clients won't prompt for passphrases. Instead, they communicate with the agent as needed. The effects of the agent last until you terminate the agent, usually just before logging out. The agent program for SSH1, SSH2, and OpenSSH is called `ssh-agent`.

Generally, you run a single `ssh-agent` in your local login session, before running any SSH clients. You can run the agent by hand, but people usually edit their login files (for example, `~/.login` or `~/.xsession`) to run the agent automatically. SSH clients communicate with the agent via the
process environment,[7] so all clients (and all other processes) within your login session have access to the agent. To try the agent, type:

```
$ ssh-agent $SHELL
```

where SHELL is the environment variable containing the name of your login shell. Alternatively, you could supply the name of any other shell, such as `sh`, `bash`, `csh`, `tcsh`, or `ksh`. The agent runs and then invokes the given shell as a child process. The visual effect is simply that another shell prompt appears, but this shell has access to the agent.

Once the agent is running, it's time to load private keys into it using the `ssh-add` program. By default, `ssh-add` loads the key from your default identity file:

```
$ ssh-add
Need passphrase for /u/you/.ssh/identity ('Your Name <you@local.org>').
Enter passphrase: ************
Identity added: /u/you/.ssh/identity ('Your Name <you@local.org>').
```

Now `ssh` and `scp` can connect to remote hosts without prompting for your passphrase. Figure 2-3 shows the process.

```
Figure 2.3. How the SSH agent works
```

`ssh-add` reads the passphrase from your terminal by default or optionally from standard input noninteractively. Otherwise, if you are running the X Window System with the DISPLAY environment variable set, and standard input isn't a terminal, `ssh-add` reads your passphrase using a graphical X program, `ssh-askpass`. This behavior is useful when calling `ssh-add` from X session setup scripts.[8]

[8] To force `ssh-add` to use X to read the passphrase, type `ssh-add < /dev/null` at a command line.

`ssh-add` has further capabilities, particularly in SSH2, and can operate with multiple identity files. [Section 6.3.3] For now, here are a few useful commands. To load a key other than your default identity into the agent, provide the filename as an argument to `ssh-add`:

```
$ ssh-add my-other-key-file
```
You can also list the keys the agent currently holds:

```
$ ssh-add -l
```

delete a key from the agent:

```
$ ssh-add -d name-of-key-file
```
or delete all keys from the agent:

```
$ ssh-add -D
```

When running an SSH agent, don't leave your terminal unattended while logged in. While your private keys are loaded in an agent, anyone may use your terminal to connect to any remote accounts accessible via those keys, without needing your passphrase! Even worse, a sophisticated intruder can extract your keys from the running agent and steal them.

If you use an agent, make sure to lock your terminal if you leave it while logged in. You can also use `ssh-add -D` to clear your loaded keys and reload them when you return. In addition, `ssh-agent2` has a "locking" feature that can protect it from unauthorized users. [Section 6.3.3]

### 2.5.1 Other Uses For Agents

Because `ssh` and `rsh` command lines have such similar syntax, you naturally might want to replace `rsh` with `ssh`. Suppose you have an automation script that uses `rsh` to run remote processes. If you use `ssh` instead, your script prompts for passphrases, which is inconvenient for automation. If the script runs `ssh` many times, retyping that passphrase repeatedly is both annoying and error-prone. If you run an agent, however, the script can run without a single passphrase prompt. [Section 11.1]

### 2.5.2 A More Complex Passphrase Problem

In our running example, we copied a file from the remote to the local host:

```
$ scp pat@shell.isp.com:print-me imprime-moi
```

In fact, `scp` can copy a file from remote host `shell.isp.com` directly to a third host running SSH on which you have an account named, say, "psmith":

```
$ scp pat@shell.isp.com:print-me psmith@other.host.net:imprime-moi
```

Rather than copying the file first to the local host and then back out again to the final destination, this command has `shell.isp.com` send it directly to `other.host.net`. However, if you try this, you will run into the following problem:

```
$ scp pat@shell.isp.com:print-me psmith@other.host.net:imprime-moi
Enter passphrase for RSA key 'Your Name <you@local.org>':
************
You have no controlling tty and no DISPLAY. Cannot read passphrase. lost connection
```

What happened? When you run `scp` on your local machine, it contacts `shell.isp.com` and internally invokes a second `scp` command to do the copy. Unfortunately, the second `scp` command also
needs the passphrase for your private key. Since there is no terminal session to prompt for the passphrase, the second `scp` fails, causing the original `scp` to fail. The SSH agent solves this problem: the second `scp` command simply queries your local SSH agent, so no passphrase prompting is needed.

The SSH agent also solves another more subtle problem in this example. Without the agent, the second `scp` (on `shell.isp.com`) needs access to your private key file, but the file is on your local machine. So you have to copy your private key file to `shell.isp.com`. This isn’t ideal; what if `shell.isp.com` isn’t a secure machine? Also, the solution doesn’t scale: if you have a dozen different accounts, it is a maintenance headache to keep your private key file on all of them. Fortunately, the SSH agent comes to the rescue once again. The remote `scp` process simply contacts your local SSH agent, authenticates, and the secure copy proceeds successfully, through a process called agent forwarding.

### 2.5.3 Agent Forwarding

In the preceding example, the remote instance of `scp` has no direct access to your private key, since the agent is running on the local host, not the remote. SSH provides *agent forwarding* [Section 6.3.5](#) to address this problem.

When agent forwarding is turned on,[9] the remote SSH server masquerades as a second `ssh-agent` as shown in Figure 2-4. It takes authentication requests from your SSH client processes there, passes them back over the SSH connection to the local agent for handling, and relays the results back to the remote clients. In short, remote clients transparently get access to the local `ssh-agent`. Since any programs executed via `ssh` on the remote side are children of the server, they all have access to the local agent just as if they were running on the local host.

[9] It is on by default in SSH1 and SSH2, but off in OpenSSH.

![Figure 2.4. How agent forwarding works](image)

In our double-remote `scp` example, here is what happens when agent forwarding comes into play (see Figure 2-5):

1. You run the command on your local machine:

   ```
   $ scp pat@shell.isp.com:print-me psmith@other.host.net:imprime-moi
   ```
2. This `scp` process contacts your local agent and authenticates you to `shell.isp.com`.
3. A second `scp` command is automatically launched on `shell.isp.com` to carry out the copy to `other.host.net`.
4. Since agent forwarding is turned on, the SSH server on `shell.isp.com` poses as an agent.
5. The second `scp` process tries to authenticate you to `other.host.net` by contacting the "agent" that is really the SSH server on `shell.isp.com`.
6. Behind the scenes, the SSH server on `shell.isp.com` communicates with your local agent, which constructs an authenticator proving your identity and passes it back to the server.
7. The server verifies your identity to the second `scp` process, and authentication succeeds on `other.host.net`.
8. The file copying occurs.

![Figure 2.5. Third-party scp with agent forwarding](image)

Agent forwarding works over multiple connections in a series, allowing you to `ssh` from one machine to another, and then to another, with the agent connection following along the whole way. These machines may be progressively less secure, but since agent forwarding doesn't send your private key to the remote host but rather relays authentication requests back to the first host for processing, your key stays safe.

### 2.6 Connecting Without a Password or Passphrase

One of the most frequently asked questions about SSH is: "How can I connect to a remote machine without having to type a password or passphrase?" As you've seen, an SSH agent can make this possible, but there are other methods as well, each with different tradeoffs. Here we list the available methods with pointers to the sections discussing each one.

To use SSH clients for *interactive sessions* without a password or passphrase, you have several options:

- Public-key authentication with an agent [Section 2.5] [Section 6.3]
Another way to achieve password-less logins is to use an unencrypted private key with no passphrase. Although this technique can be appropriate for automation purposes, never do this for interactive use. Instead, use the SSH agent, which provides the same benefits with much greater security. Don't use unencrypted keys for interactive SSH!

On the other hand, noninteractive, unattended programs such as cron jobs or batch scripts may also benefit from not having a password or passphrase. In this case, the different techniques raise some complex issues, and we will discuss their relative merits and security issues later. [Section 11.1]

2.7 Miscellaneous Clients

Several other clients are included in addition to ssh and scp:

- sftp, an ftp-like client for SSH2
- slogin, a link to ssh, analogous to the rlogin program
- Hostname links to ssh

2.7.1 sftp

The scp command is convenient and useful, but many users are already familiar with FTP (File Transfer Protocol), a more widely used technique for transferring files on the Internet.\[10\] sftp is a separate file-transfer tool layered on top of SSH. It was developed by SSH Communications Security and was originally available only in SSH2, but other implementations have since appeared (e.g., client support in SecureFX and server support in OpenSSH). sftp is available only in SSH2: it is implemented as an SSH2 subsystem [Section 5.7] and thus not readily adaptable to use with SSH1.

\[10\] Due to the nature of the FTP protocol, FTP clients are difficult to secure using TCP port forwarding, unlike most other TCP-based clients. [Section 11.2]

sftp is advantageous for several reasons:

- It is secure, using an SSH-protected channel for data transfer.
- Multiple commands for file copying and manipulation can be invoked within a single sftp session, whereas scp opens a new session each time it is invoked.
- It can be scripted using the familiar ftp command language.
- In other software applications that run an FTP client in the background, you can try substituting sftp, thus securing the file transfers of that application.

You may need an agent when trying this or similar FTP replacements, since programs that use FTP might not recognize the prompt sftp issues for your passphrase, or they might expect you to have suppressed FTP's password prompt (using a .netrc file, for example).

Anyone familiar with FTP will feel right at home with sftp, but sftp has some additional features of note:
• Command-line editing using GNU Emacs-like keystrokes (Control-B for backward character, Control-E for end of line, and so forth)
• Regular-expression matching for filenames, as documented in the sshregex manpage supplied with SSH2 and found in Appendix A
• Several command-line options:

-b filename
Read commands from the given file instead of the terminal

-S path
Locate the ssh2 program using the given path

-h
Print a help message and exit

-V
Print the program version number and exit

-D module=level
Print debugging output [Section 5.8.2.2]

Also, sftp doesn't have the separate ASCII and binary transfer modes of standard FTP, only binary. All files are transferred literally. Therefore, if you copy ASCII text files between Windows and Unix with sftp, end-of-line characters aren't translated properly. Normally, FTP's ASCII mode translates between Windows' "carriage return plus newline" and Unix's newline, for example.

2.7.2 slogin

slogin is an alternative name for ssh, just as rlogin is a synonym for rsh. On Unix systems, slogin is simply a symbolic link to ssh. Note that the slogin link is found in SSH1 and OpenSSH but not SSH2. We recommend using just ssh for consistency: it's found in all these implementations and is shorter to type.

2.7.3 Hostname Links

ssh for SSH1 and OpenSSH also mimics rlogin in another respect: support for hostname links. If you make a link to the ssh executable, and the link name isn't in the set of standard names ssh recognizes,[11] ssh has special behavior. It treats the link name as a hostname and attempts to connect to that remote host. For example, if you create a link called terpsichore.muses.org and then run it:

[11] These are rsh, ssh, rlogin, slogin, ssh1, slogin1, ssh.old, slogin.old, ssh1.old, slogin1.old, and remsh.

$ ln -s /usr/local/bin/ssh terpsichore.muses.org
$ terpsichore.muses.org
Welcome to Terpsichore!  Last login January 21st, 201 B.C.
terpsichore>
It's equivalent to running:

$ ssh terpsichore.muses.org
Welcome to Terpsichore! Last login January 21st, 201 B.C.
terpsichore>

You can create a collection of these links for all commonly used remote hosts. Note that support for hostname links has been removed in SSH2. (We have never found them to be very useful, ourselves, but the capability does exist in SSH1 and OpenSSH.)

2.8 Summary

From the user's point of view, SSH consists of several client programs and some configuration files. The most commonly used clients are ssh for remote login and scp for file transfer. Authentication to the remote host can be accomplished using existing login passwords or with public-key cryptographic techniques. Passwords are more immediately and easily used, but public-key authentication is more flexible and secure. The ssh-keygen, ssh-agent, and ssh-add programs generate and manage SSH keys.
Chapter 3. Inside SSH

SSH secures your data while it passes over a network, but how exactly does it work? In this chapter, we move firmly onto technical ground and explain the inner workings of SSH. Let's roll up our sleeves and dive into the bits and bytes.

This chapter is written for system administrators, network administrators, and security professionals. Our goal is to teach you enough about SSH to make an intelligent, technically sound decision about using it. We cover the SSH-1 and SSH-2 protocols separately since they have important differences.

Of course, the ultimate references on SSH are the protocol standards and the source code of an implementation. We don't completely analyze the protocols or recapitulate every step taken by the software. Rather, we summarize them to provide a solid, technical overview of their operation. If you need more specifics, you should refer to the standards documents. The SSH Version 2 protocol is in draft status on the IETF standards track; it is available at:

http://www.ipsec.com/tech/archive/secsh.html
http://www.ietf.org/

The older protocol implemented in SSH1 and OpenSSH/1 is Version 1.5 and is documented in a file named RFC included in the SSH1 source package.

3.1 Overview of Features

The major features and guarantees of the SSH protocol are:

- **Privacy** of your data, via strong encryption
- **Integrity** of communications, guaranteeing they haven't been altered
- **Authentication**, i.e., proof of identity of senders and receivers
- **Authorization**, i.e., access control to accounts
- **Forwarding or tunneling** to encrypt other TCP/IP-based sessions

3.1.1 Privacy (Encryption)

Privacy means protecting data from disclosure. Typical computer networks don't guarantee privacy; anyone with access to the network hardware, or to hosts connected to the network may be able to read (or sniff) all data passing over the network. Although modern switched networks have reduced this problem in local area networks, it is still a serious issue; passwords are regularly stolen by such sniffing attacks.

SSH provides privacy by encrypting data that passes over the network. This end-to-end encryption is based on random keys that are securely negotiated for that session and then destroyed when the session is over. SSH supports a variety of encryption algorithms for session data, including such standard ciphers as ARCFOUR, Blowfish, DES, IDEA, and triple-DES (3DES).

3.1.2 Integrity

Integrity means assuring that data transmitted from one end of a network connection arrives unaltered on the other side. The underlying transport of SSH, TCP/IP, does have integrity
checking to detect alteration due to network problems (electrical noise, lost packets due to excessive traffic, etc.). Nevertheless, these methods are ineffective against deliberate tampering and can be fooled by a clever attacker. Even though SSH encrypts the data stream so an attacker can't easily change selected parts to achieve a specific result, TCP/IP's integrity checking alone can't prevent, say, an attacker's deliberate injection of garbage into your session.

A more complex example is a **replay attack**. Imagine that Attila the Attacker is monitoring your SSH session and also simultaneously watching over your shoulder (either physically, or by monitoring your keystrokes at your terminal). In the course of your work, Attila sees you type the command `rm -rf *` within a small directory. He can't read the encrypted SSH session data, of course, but he could correlate a burst of activity on that connection with your typing the command and capture the packets containing the encrypted version of your command. Later, when you're working in your home directory, Attila inserts the captured bits into your SSH session, and your terminal mysteriously erases all your files!

Attila's replay attack succeeds because the packets he inserted are valid; he could not have produced them himself (due to the encryption), but he can copy and replay them later. TCP/IP's integrity check is performed only on a per-packet basis, so it can't detect Attila's attack. Clearly, the integrity check must apply to the data stream as a whole, ensuring that the bits arrive as they were sent: in order and with no duplication.

The SSH-2 protocol uses cryptographic integrity checking, which verifies both that transmitted data hasn't been altered and that it truly comes from the other end of the connection. SSH-2 uses keyed hash algorithms based on MD5 and SHA-1 for this purpose: well known and widely trusted algorithms. SSH-1, on the other hand, uses a comparatively weak method: a 32-bit cyclic redundancy check (CRC-32) on the unencrypted data in each packet. [Section 3.9.3](#)

### 3.1.3 Authentication

Authentication means verifying someone's identity. Suppose I claim to be Richard Silverman, and you want to authenticate that claim. If not much is at stake, you might just take my word for it. If you're a little concerned, you might ask for my driver's license or other photo ID. If you're a bank officer deciding whether to open a safe-deposit box for me, you might also require that I possess a physical key, and so on. It all depends on how sure you want to be. The arsenal of high-tech authentication techniques is growing constantly and includes DNA-testing microchips, retina and hand scanners, and voice-print analyzers.

Every SSH connection involves two authentications: the client verifies the identity of the SSH server (**server authentication**), and the server verifies the identity of the user requesting access (**user authentication**). Server authentication ensures that the SSH server is genuine, not an impostor, guarding against an attacker's redirecting your network connection to a different machine. Server authentication also protects against man-in-the-middle attacks, wherein the attacker sits invisibly between you and the server, pretending to be the client on one side and the server on the other, fooling both sides and reading all your traffic in the process!

There is difference of opinion as to the granularity of server authentication: should it be distinguish between different server hosts, or between individual instances of the SSH server? That is, must all SSH servers running on a particular host have the same host key, or might they have different ones? The term "host key," of course, reflects a bias towards the first interpretation, which SSH1 and OpenSSH follow: their known-hosts lists can only associate a single key with any particular hostname. SSH2, on the other hand, uses the second approach: "host keys" are actually associated with individual listening sockets, allowing multiple keys per host. This may reflect a pragmatic need rather than a considered change in principle. When SSH2 first appeared, it supported only DSA host keys, whereas SSH-1 supports only RSA keys. It was therefore impossible, as a matter of implementation, for a single host to run both SSH-1 and SSH2 servers and have them share a host key.
User authentication is traditionally done with passwords, which unfortunately are a weak authentication scheme. To prove your identity you have to reveal the password, exposing it to possible theft. Additionally, in order to remember a password, people are likely to keep it short and meaningful, which makes the password easier for third parties to guess. For longer passwords, some people choose words or sentences in their native languages, and these passwords are likely to be crackable. From the standpoint of information theory, grammatical sentences contain little real information (technically known as entropy): generally less than two bits per character in English text, far less than the 8 - 16 bits per character found in computer encodings.

SSH supports authentication by password, encrypting the password as it travels over the network. This is a vast improvement over other common remote-access protocols (Telnet, FTP) which generally send your password in the clear (i.e., unencrypted) over the network, where anyone with sufficient network access can steal it! Nevertheless, it's still only simple password authentication, so SSH provides other stronger and more manageable mechanisms: per-user public-key signatures, and an improved rlogin-style authentication, with host identity verified by public key. In addition, various SSH implementations support some other systems, including Kerberos, RSA Security's SecurID tokens, S/Key one-time passwords, and the Pluggable Authentication Modules (PAM) system. An SSH client and server negotiate to determine which authentication mechanism to use, based on their configurations. SSH2 can even require multiple forms of authentication.

### 3.1.4 Authorization

Authorization means deciding what someone may or may not do. It occurs after authentication, since you can't grant someone privileges until you know who she is. SSH servers have various ways of restricting clients' actions. Access to interactive login sessions, TCP port and X window forwarding, key agent forwarding, etc., can all be controlled, though not all these features are available in all SSH implementations, and they aren't always as general or flexible as you might want. Authorization may be controlled at a serverwide level (e.g., the `/etc/ssh_config` file for SSH1), or per account, depending on the authentication method used (e.g., each user's files `~/.ssh/authorized_keys`, `~/.ssh2/authorization`, `~/.shosts`, `~/.k5login`, etc.).

### 3.1.5 Forwarding (Tunneling)

Forwarding or tunneling means encapsulating another TCP-based service, such as Telnet or IMAP, within an SSH session. This brings the security benefits of SSH (privacy, integrity, authentication, authorization) to other TCP-based services. For example, an ordinary Telnet connection transmits your username, password, and the rest of your login session in the clear. By forwarding `telnet` through SSH, all of this data is automatically encrypted and integrity-checked, and you may authenticate using SSH credentials.

SSH supports three types of forwarding. General TCP port forwarding operates as described earlier for any TCP-based service. [Section 9.2] X forwarding comprises additional features for securing the X protocol (i.e., X windows). [Section 9.3] The third type, agent forwarding, permits SSH clients to access SSH public keys held on remote machines. [Section 6.3.5]

### 3.2 A Cryptography Primer

We've covered the basic properties of SSH. Now we focus on cryptography, introducing important terms and ideas regarding the technology in general. There are many good references on cryptographic theory and practice, and we make no attempt here to be comprehensive. (For more detailed information, check out Bruce Schneier's excellent book, *Applied Cryptography*, published by John Wiley & Sons.) We introduce encryption and decryption, plaintext and ciphertext, keys,
secret-key and public-key cryptography, and hash functions, both in general and as they apply to SSH.

Encryption is the process of scrambling data so that it can't be read by unauthorized parties. An encryption algorithm (or cipher) is a particular method of performing the scrambling; examples of currently popular encryption algorithms are RSA, RC4, DSA, and IDEA. The original, readable data is called the plaintext, or data in the clear, while the encrypted version is called the corresponding ciphertext. To convert plaintext to ciphertext, you apply an encryption algorithm parameterized by a key, a string that is typically known only to you. An encryption algorithm is considered secure if it is infeasible for anyone to read (or decrypt) the encrypted data without the key. An attempt to decrypt data without its key is called cryptanalysis.

3.2.1 How Secure Is Secure?

It's important to understand the word "infeasible" in the previous paragraph. Today's most popular and secure ciphers are vulnerable to brute-force attacks: if you try every possible key, you will eventually succeed in decryption. However, when the number of possible keys is large, a brute-force search requires a great deal of time and computing power. Based on the state of the art in computer hardware and algorithms, it is possible to pick sufficiently large key sizes so as to render brute-force key search infeasible for your adversary. What counts as infeasible, though, varies depending on how valuable the data is, how long it must stay secure, and how motivated and well-funded your adversary is. Keeping something secret from your rival startup for a few days is one thing; keeping it secret from a major world government for 10 years is quite another.

Of course, for all this to make sense, you must be convinced that brute force is the only way to attack your cipher. Encryption algorithms have structure and are susceptible to mathematical analysis. Over the years, many ciphers previously thought secure have fallen to advances in cryptanalysis. It isn't currently possible to prove a practical cipher secure. Rather, a cipher acquires respectability through intensive study by mathematicians and cryptographers. If a new cipher exhibits good design principles, and well-known researchers study it for some time and fail to find a practical, faster method of breaking it than brute force, then people will consider it secure.[1]

[1] In his pioneering works on information theory and encryption, the mathematician Claude Shannon defined a model for cipher security and showed there is a cipher that is perfectly secure under that model: the so-called one-time pad. It is perfectly secure: the encrypted data gives an attacker no information whatsoever about the possible plaintexts. The ciphertext literally can decrypt to any plaintext at all with equal likelihood. The problem with the one-time pad is that it cumbersome and fragile. It requires that keys be as large as the messages they protect, be generated perfectly randomly, and never be reused. If any of these requirements are violated, the one-time pad becomes extremely insecure. The ciphers in common use today aren't perfectly secure in Shannon's sense, but for the best of them, brute-force attacks are infeasible.

3.2.2 Public- and Secret-Key Cryptography

Encryption algorithms as described so far are called symmetric or secret-key ciphers; the same key is used for encrypting and decrypting. Examples are Blowfish, DES, IDEA, and RC4. Such a cipher immediately introduces the key-distribution problem: how do you get the key to your intended recipient? If you can meet in person every once and a while and exchange a list of keys, all well and good, but for dynamic communication over computer networks, this doesn't work.

Public-key, or asymmetric, cryptography replaces the single key with a pair of related keys: public and private. They are related in a mathematically clever way: data encrypted with the public key may be decrypted with its private counterpart, and it is infeasible to derive the private key from the public one. You keep your private key, well... private, and give the public key to anyone who wants it, without worrying about disclosure. Ideally, you publish it in a directory next to your name, like a telephone book. When someone wants to send you a secret message, they encrypt it
with your public key. Other people may have your public key, but that won't allow them to
decrypt the message; only you can do that with the corresponding private key. Public-key
cryptography goes a long way towards solving the key-distribution problem.\footnote{There is still the issue of reliably determining whose public key is whose; but that gets into public-key infrastructure, or PKI systems, and is a broader topic.}

Public-key methods are also the basis for \textit{digital signatures}: extra information attached to a digital
document to provide evidence that a particular person has seen and agreed to it, much as a pen-and-ink signature does with a paper document. Any asymmetric cipher (RSA, ElGamal, Elliptic Curve, etc.) may be used for digital signatures, though the reverse isn't true. For instance, the DSA algorithm, which is used by the SSH-2 protocol for its keys, is a signature-only public-key scheme and can't be used for encryption.\footnote{That's the idea, anyway, although it has been pointed out that it's easy to use a general DSA implementation for both RSA and ElGamal encryption. That was not the intent, however.}

Secret- and public-key encryption algorithms differ in another way: performance. All common
public-key algorithms are enormously slower than secret-key ciphers—by orders of magnitude. It
is simply infeasible to encrypt large quantities of data using a public-key cipher. For this reason,
modern data encryption uses both methods together. Suppose you want to send some data securely
to your friend Bob Bitflipper. Here's what a modern encryption program does:

1. Generate a random key, called the \textit{bulk key}, for a fast, secret-key algorithm such as 3 DES
   (a.k.a the \textit{bulk cipher}).
2. Encrypt the plaintext with the bulk key.
3. Secure the bulk key by encrypting it with Bob Bitflipper's public key, so only Bob can
decrypt it. Since secret keys are small (a few hundred bits long at most), the speed of the
   public-key algorithm isn't an issue.

To reverse the operation, Bob's decryption program first decrypts the bulk key, and then uses it to
decrypt the ciphertext. This method yields the advantages of both kinds of encryption technology,
and in fact, SSH uses this technique. User data crossing an SSH connection is encrypted using a
fast secret-key cipher, the key for which is shared between the client and server using public-key
methods.

\subsection{3.2.3 Hash Functions}

In cryptography (and elsewhere in computing and network technology), it is often useful to know
if some collection of data has changed. Of course, one can just send along (or keep around) the
original data for comparison, but that can be prohibitively expensive both in time and storage. The
common tool addressing this need is called a \textit{hash function}. Hash functions are used by SSH-1 for
integrity checking (and have various other uses in cryptography we won't discuss here).

A hash function is simply a mapping from a larger set of data values to a smaller set. For instance,
a hash function \( H \) might take an input bit string of any length up to 50,000 bits, and uniformly
produce a 128-bit output. The idea is that when sending a message \( m \) to Alice, I also send along
the hash value \( H(m) \). Alice computes \( H(m) \) independently and compares it to the \( H(m) \) value I sent;
if they differ, she concludes that the message was modified in transit.

This simple technique can't be completely effective. Since the range of the hash function is strictly
smaller than its domain, many different messages have the same hash value. To be useful, \( H \) must
have the property that the kinds of alterations expected to happen to the messages in transit, must
be overwhelmingly likely to cause a change in the message hash. Put another way: given a
message \( m \) and a typical changed message \( m' \), it must be extremely unlikely that \( H(m) = H(m') \).
Thus a hash function must be tailored to its intended use. One common use is in networking: datagrams transmitted over a network frequently include a message hash that detects transmission errors due to hardware failure or software bugs. Another use is in cryptography, to implement digital signatures. Signing a large amount of data is prohibitively expensive, since it involves slow public-key operations as well as shipping along a complete encrypted copy of the data. What is actually done is to first hash the document, producing a small hash value, and then sign that, sending the signed hash along instead. A verifier independently computes the hash, then decrypts the signature using the appropriate public key, and compares them. If they are the same, he concludes (with high probability) that the signature is valid, and that the data hasn’t changed since the private key holder signed it.

These two uses, however, have different requirements, and a hash function suitable for detecting transmission errors due to line noise might be ineffective at detecting deliberate alterations introduced by a human attacker! A cryptographic hash function must make it computationally infeasible to find two different messages having the same hash or to find a message having a particular fixed hash. Such a function is said to be collision-resistant (or collision-proof; though that’s a bit misleading), and pre-image-resistant. The Cyclic Redundancy Check hash commonly used to detect accidental data changes (e.g., in Ethernet frame transmissions) is an example of a non-collision-resistant hash. It is easy to find CRC-32 hash collisions, and the SSH-1 insertion attack is based on this fact. [Section 3.10.5] Examples of cryptographically strong hash functions are MD5 and SHA-1.

3.3 The Architecture of an SSH System

SSH has about a dozen distinct, interacting components that produce the features we’ve covered. [Section 3.1] Figure 3-1 illustrates the major components and their relationships to one another.

Figure 3.1. SSH architecture

By "component" we don't necessarily mean "program:" SSH also has keys, sessions, and other fun things. In this section we provide a brief overview of all the components, so you can begin to get the big picture of SSH:

Server
A program that allows incoming SSH connections to a machine, handling authentication, authorization, and so forth. In most Unix SSH implementations, the server is `sshd`.

**Client**

A program that connects to SSH servers and makes requests, such as "log me in" or "copy this file." In SSH1, SSH2, and OpenSSH, the major clients are `ssh` and `scp`.

**Session**

An ongoing connection between a client and a server. It begins after the client successfully authenticates to a server and ends when the connection terminates. Sessions may be interactive or batch.

**Key**

A relatively small amount of data, generally from tens to one or two thousand bits, used as a parameter to cryptographic algorithms such as encryption or message authentication. The use of the key binds the algorithm operation in some way to the key holder: in encryption, it ensures that only someone else holding that key (or a related one) can decrypt the message; in authentication, it allows you to later verify that the key holder actually signed the message. There are two kinds of keys: symmetric or secret-key, and asymmetric or public-key. [Section 3.2.2] An asymmetric key has two parts: the public and private components. SSH deals with four types of keys, as summarized in Table 3-1 and described following the table.

<table>
<thead>
<tr>
<th>Name</th>
<th>Lifetime</th>
<th>Created by</th>
<th>Type</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>User key</td>
<td>Persistent</td>
<td>User</td>
<td>Public</td>
<td>Identify a user to the server</td>
</tr>
<tr>
<td>Session key</td>
<td>One session</td>
<td>Client (and server)</td>
<td>Secret</td>
<td>Protect communications</td>
</tr>
<tr>
<td>Host key</td>
<td>Persistent</td>
<td>Administrator</td>
<td>Public</td>
<td>Identify a server/machine</td>
</tr>
<tr>
<td>Server key</td>
<td>One hour</td>
<td>Server</td>
<td>Public</td>
<td>Encrypt the session key (SSH1 only)</td>
</tr>
</tbody>
</table>

**User key**

A persistent, asymmetric key used by clients as proof of a user's identity. (A single user may have many keys/identities.)

**Host key**

A persistent, asymmetric key used by a server as proof of its identity, as well as by a client when proving its host's identity as part of trusted-host authentication. [Section 3.4.2.3] If a machine runs a single SSH server, the host key also uniquely identifies the machine. (If a machine is running multiple SSH servers, each may have a different host key, or they may share.) Often confused with the server key.

**Server key**

A temporary, asymmetric key used in the SSH-1 protocol. It is regenerated by the server at regular intervals (by default every hour) and protects the session key (defined shortly). Often confused with the host key. This key is never explicitly stored on disk, and its private component is never transmitted over the connection in any form; it provides "perfect forward secrecy" for SSH-1 sessions. [Section 3.4.1]
Session key

A randomly generated, symmetric key for encrypting the communication between an SSH client and server. It is shared by the two parties in a secure manner during the SSH connection setup, so that an eavesdropper can't discover it. Both sides then have the session key, which they use to encrypt their communications. When the SSH session ends, the key is destroyed.

SSH-1 uses a single session key, but SSH-2 has several: each direction (server to client, and client to server) has keys for encryption and others for integrity checking. In our discussions we treat all SSH-2's session keys as a unit and speak of "the session key" for convenience. If the context requires it, we specify which individual key we mean.

Key generator

A program that creates persistent keys (user keys and host keys) for SSH. SSH1, SSH2, and OpenSSH have the program ssh-keygen.

Known hosts database

A collection of host keys. Clients and servers refer to this database to authenticate one another.

Agent

A program that caches user keys in memory, so users needn't keep retyping their passphrases. The agent responds to requests for key-related operations, such as signing an authenticator, but it doesn't disclose the keys themselves. It is a convenience feature. SSH1, SSH2, and OpenSSH have the agent ssh-agent, and the program ssh-add loads and unloads the key cache.

Signer

A program that signs hostbased authentication packets. We explain this in our discussion of trusted-host authentication. [Section 3.4.2.3]

Random seed

A pool of random data used by SSH components to initialize software pseudo-random number generators.

Configuration file

A collection of settings to tailor the behavior of an SSH client or server.

Not all these components are required in an implementation of SSH. Certainly servers, clients, and keys are mandatory, but many implementations don't have an agent, and some even don't include a key generator.

3.4 Inside SSH-1
Now that we’ve seen the major features and components of SSH, let’s delve into the details of the SSH-1 protocol. SSH-2 is covered separately. [Section 3.5] The architecture of SSH-1 is summarized in Figure 3-2. We will cover:

- How the secure session is established
- Authentication by password, public key, or trusted host
- Integrity checking
- Data compression

**Figure 3.2. SSH-1 architecture**

![SSH-1 Architecture Diagram]

### 3.4.1 Establishing the Secure Connection

Before meaningful interaction can take place, the SSH client and server must establish a secure connection. This lets them share keys, passwords, and ultimately, whatever data they transmit to each other.

We will now explain how the SSH-1 protocol guarantees security of a network connection. Through a multistep process, starting from scratch, the SSH-1 client and server agree on an encryption algorithm and generate and share a secret session key, establishing a secure connection:

1. The client contacts the server.
2. The client and server disclose the SSH protocol versions they support.
3. The client and server switch to a packet-based protocol.
4. The server identifies itself to the client and provides session parameters.
5. The client sends the server a secret (session) key.
6. Both sides turn on encryption and complete server authentication.
7. The secure connection is established.
Now the client and server can communicate by encrypted messages. Let’s examine each step in
detail; the complete process is summarized in Figure 3-3.

Figure 3.3. SSH-1 protocol exchange

1. **The client contacts the server.**

   This is done without fanfare, simply by sending a connection request to the server’s TCP
   port, which is port 22 by convention.

2. **The client and server disclose the SSH protocol versions they support.**

   These protocols are represented as ASCII strings, such as "SSH-1.5-1.2.27", which means
   SSH protocol Version 1.5 as implemented by SSH1 Version 1.2.27. You can see this
   string by connecting to an SSH server port with a Telnet client:

   ```
   $ telnet server 22
   Trying 192.168.10.1
   Connected to server (192.168.10.1).
   Escape character is '^]'.
   SSH-1.5-1.2.27
   ```
The implementation version (1.2.27) is just a comment and is optional in the string. But, some implementations examine the comment to recognize particular software versions and work around known bugs or incompatibilities.\footnote{Some system administrators remove the comment, preferring not to announce their software package and version to the world, which provides clues to an attacker.}

If the client and server decide their versions are compatible, the connection process continues; otherwise either party may decide to terminate the connection. For instance, if an SSH-1-only client encounters an SSH-2-only server, the client disconnects and prints an error message. Other actions are possible: for example, the SSH-2-only server can invoke an SSH-1 server to handle the connection.

3. **The client and server switch to a packet-based protocol.**

Once the protocol version exchange is complete, both sides switch to a packet-based protocol over the underlying TCP connection. Each packet consists of a 32-bit length field, 1-8 bytes of random padding to foil known-plaintext attacks, a one-byte packet type code, the packet payload data, and a four-byte integrity check field.

4. **The server identifies itself to the client and provides session parameters.**

The server sends the following information to the client (all still unencrypted):

- Its host key, used to prove the server host identity later.
- Its server key, which helps establish the secure connection.
- A sequence of eight random bytes, called check bytes. The client must include these check bytes in its next response, or the server rejects the response. This measure protects against some IP spoofing attacks.
- Lists of encryption, compression, and authentication methods that the server supports.

At this point, both sides also compute a common 128-bit session identifier, which is used in some subsequent protocol operations to uniquely identify this SSH session. This is an MD5 hash of the host key, server key, and check bytes taken together.

When the client receives the host key, it asks the question: “Have I spoken with this server before, and if so, what was its host key then?” To answer this question, the client consults its known hosts database. If the newly arrived host key matches a previous one in the database, all is well. However, there are two other possibilities: the server might not appear in the known hosts database, or it might be present but with a different host key. In each of these cases, the client elects to trust the newly arrived key or to reject it. \[Section 7.4.3.1\] Human guidance may be needed: for example, the client's user can be prompted to accept or reject the key.

If the client rejects the host key, the connection ends. Let's assume the host key is acceptable and continue.

5. **The client sends the server a secret (session) key.**

Now the client randomly generates a new key for a bulk cipher \[Section 3.2.2\] that both client and server support; this is called the session key. Its purpose is to encrypt and decrypt messages sent between the client and the server. All that's needed is to give this session key to the server, and both sides can turn on encryption and begin communicating securely.
Of course, the client can't simply send the session key to the server. Encryption isn't operating yet, and if a third party intercepts this key, it can decrypt the client's and server's messages. Goodbye security! So the client must send the session key securely. This is done by encrypting it twice: once with the server's public host key and once with the server key. This step ensures that only the server can read it. After the session key is double-encrypted, the client sends it to the server, along with the check bytes and a choice of algorithms (picked from the server's list of supported algorithms sent in Step 4).

6. *Both sides turn on encryption and complete server authentication.*

After sending the session key, both sides begin encrypting the session data with the key and the selected bulk cipher. Before sending anything further, though, the client waits for a confirmation message from the server, which (like all subsequent data) must be encrypted with the session key. This final step provides the server authentication: only the intended server can have decrypted the session key, since it was encrypted with the host key verified earlier against the known hosts list.

Without the session key, an impostor server can't decrypt the subsequent protocol traffic or produce valid traffic in return, and the client will notice and terminate the connection.

Note that server authentication is implicit; there's no explicit exchange to verify the server host key. Therefore it's important for the client to wait for a valid server response using the new session key before sending anything further, in order to verify the server's identity before proceeding. The SSH-1 protocol isn't specific about this point, but SSH-2 requires it when server authentication is implicit in the session key exchange.

Encrypting the session key a second time with the server key provides a property called **perfect forward secrecy**. This means there are no persistent keys lying around whose disclosure can jeopardize the secrecy of past or future SSH sessions. If the server host key alone is used to protect the session key, then disclosure of the host private key compromises future communications and allows decryption of old, recorded sessions. Using the server key in tandem for this purpose removes this weakness, as it is temporary, never explicitly stored on disk, and replaced periodically (by default, once an hour). Having stolen the server private key, an interloper must still perform an active man-in-the-middle or server spoofing attack to compromise a session.

7. *The secure connection is established.*

Since both the client and server now know the session key, and nobody else does, they can send each other encrypted messages (using the bulk cipher they agreed on) only they can decrypt. Also, the client has completed server authentication. We're ready to begin client authentication.

### 3.4.2 Client Authentication

Once the secure connection is established, the client attempts to authenticate itself to the server. The client may try any authentication methods at its disposal until one succeeds, or all have failed. For example, the six authentication methods defined by the SSH-1.5 protocol, in the order attempted by the SSH1 implementation, are:


   [5] This method isn't available by default; it must be requested at compile time.

2. Rhosts
3. RhostsRSA
4. Public-key
5. TIS
6. Password (flavors: host login password, Kerberos, SecurID, S/Key, etc.)

F-Secure SSH Client for Windows (see Chapter 16) tries these in order:
1. Public-key
2. Password

Knowing the order for your client is a good idea. It helps to diagnose problems when authentication fails or acts unexpectedly.

3.4.2.1 Password authentication

During password authentication, the user supplies a password to the SSH client, which the client transmits securely to the server over the encrypted connection. The server then checks that the given password is acceptable for the target account, and allows the connection if so. In the simplest case, the SSH server checks this through the native password-authentication mechanism of the host operating system.

Password authentication is quite convenient because it requires no additional setup for the user. You don't need to generate a key, create a ~/.ssh directory on the server machine, or edit any configuration files. This is particularly convenient for first-time SSH users and for users who travel a lot and don't carry their private keys. You might not want to use your private keys on other machines, or there may be no way to get them onto the machine in question. If you frequently travel, you should consider setting up SSH to use one-time passwords if your implementation supports them, improving the security of the password scheme. [Section 3.4.2.5]

On the other hand, password authentication is inconvenient because you have to type a password every time you connect. Also, password authentication is less secure than public-key because the sensitive password is transmitted off the client host. It is protected from snooping while on the network but is vulnerable to capture once it arrives at the server if that machine has been compromised. This is in contrast with public-key authentication, as even a compromised server can't learn your private key through the protocol. Therefore, before choosing password authentication, you should weigh the trustworthiness of the client and the server, as you will be revealing to them the key to your electronic kingdom.

Password authentication is simple in concept, but different Unix variants store and verify passwords in different ways, leading to some complexities. OpenSSH uses PAM for password authentication by default, which must be carefully configured. [Section 4.3] Most Unix systems encrypt passwords with DES (via the crypt() library routine), but recently some systems have started using the MD5 hash algorithm, leading to configuration issues. [Section 4.3] The behavior of password authentication also changes if Kerberos [Section 5.5.1.7] or SecurID support [Section 5.5.1.9] is enabled in the SSH server.

3.4.2.2 Public-key authentication

Public-key authentication uses public-key cryptography to verify the client's identity. To access an account on an SSH server machine, the client proves that it possesses a secret: specifically, the private counterpart of an authorized public key. A key is "authorized" if its public component is contained in the account's authorization file (e.g., ~/.ssh/authorized_keys). The sequence of actions is:
1. The client sends the server a request for public-key authentication with a particular key. The request contains the key's modulus as an identifier.\[6\]

\[\text{An RSA key consists of two parts: the exponent and the modulus. The modulus is the long number in the public key (.pub) file.}\]

The key is implicitly RSA; the SSH-1 protocol specifies the RSA algorithm particularly and exclusively for public-key operations.

2. The server reads the target account's authorization file, and looks for an entry with the matching key. If there is no matching entry, this authentication request fails.

3. If there is a matching entry, the server retrieves the key and notes any restrictions on its use. The server can then reject the request immediately on the basis of a restriction, for example, if the key shouldn't be used from the client host. Otherwise, the process continues.

4. The server generates a random 256-bit string as a challenge, encrypts it with the client's public key, and sends this to the client.

5. The client receives the challenge and decrypts it with the corresponding private key. It then combines the challenge with the session identifier, hashes the result with MD5, and returns the hash value to the server as its response to the challenge. The session identifier is mixed in to bind the authenticator to the current session, protecting against replay attacks taking advantage of weak or compromised random-number generation in creating the challenge.

The hashing operation is there to prevent misuse of the client's private key via the protocol, including a chosen-plaintext attack.\[7\] If the client simply returns the decrypted challenge instead, a corrupt server can present any data encrypted with the client's public key, and the unsuspecting client dutifully decrypts and returns it. It might be the data-encryption key for an enciphered email message the attacker intercepted. Also, remember that with RSA, "decrypting" some data with the private key is actually the same operation as "signing" it. So the server can supply chosen, unencrypted data to the client as a "challenge," to be signed with the client's private key—perhaps a document saying, "OWAH TAGU SIAM" or something even more nefarious.

\[\text{In a chosen-plaintext attack, the cryptanalyst is allowed to examine plaintext/ciphertext pairs of her choosing, encrypted with the key she's trying to break. The RSA algorithm is particularly vulnerable to chosen-plaintext attacks, so it's important for a protocol using RSA to avoid them.}\]

6. The server computes the same MD5 hash of the challenge and session ID; if the client's reply matches, the authentication succeeds.

The public-key method is the most secure authentication option available in SSH, generally speaking. First of all, the client needs two secrets to authenticate: the private key, and the passphrase to decrypt it. Stealing either one alone doesn't compromise the target account (assuming a strong passphrase). The private key is infeasible to guess and never leaves the client host, making it more difficult to steal than a password. A strong passphrase is difficult to guess by brute force, and if necessary, you can change your passphrase without changing the associated key. Also, public-key authentication doesn't trust any information supplied by the client host; proof of possession of the private key is the sole criterion. This is in contrast to RhostsRSA authentication, in which the server delegates partial responsibility for the authentication process to the client host: having verified the client host's identity and privilege of the client running on it, it trusts the client software not to lie about the user's identity. [Section 3.4.2.3] If someone can impersonate a client host, he can impersonate any user on that host without actually having to steal anything from the user. This can't happen with public-key authentication.\[8\]
Don’t confuse impersonating the client host with compromising it, however. If you actually break into the client host and compromise its security, all bets are off; you can then steal the keys, passwords, etc., of any users on that host. SSH doesn’t protect against host compromise.

Public-key authentication is also the most flexible method in SSH for its additional control over authorization. You may tag each public key with restrictions to be applied after authentication succeeds: which client hosts may connect, what commands may be run, and so on. [Section 8.2] This isn’t an intrinsic advantage of the public-key method, of course, but rather an implementation detail of SSH, albeit an important one.[9]

We wish this were done differently. Rather than entangling the authentication and authorization functions in this way, SSH should be able to apply any restriction to any connection, regardless of the authentication method. However, no implementation of SSH, to our knowledge, keeps authentication and authorization truly orthogonal.

On the down side, public-key authentication is more cumbersome than the other methods. It requires users to generate and maintain their keys and authorization files, with all the attendant possibilities for error: syntax errors in authorized_keys entries, incorrect permissions on SSH directories or files, lost private key files requiring new keys and updates to all target accounts, etc. SSH doesn’t provide any management infrastructure for distributing and maintaining keys on a large scale. You can combine SSH with the Kerberos authentication system, which does provide such management, to obtain the advantages of both. [Section 11.4]

One technical limitation regarding public-key authentication arises in connection with the RSAref encryption library. [Section 3.9.1.1] RSAref supports key lengths only up to 1024 bits, whereas the SSH internal RSA software supports longer keys. If you try to use a longer key with SSH/RSAref, you get an error. This can happen with either user or host keys, perhaps preexisting ones if you’ve recently switched to RSAref, or keys transferred from systems running the non-RSAref version of SSH. In all these cases, you have to replace the keys with shorter ones.

3.4.2.3 Trusted-host authentication (Rhosts and RhostsRSA)

Password and public-key authentication require the client to prove its identity by knowledge of a secret: a password or a private key particular to the target account on the server. In particular, the client’s location—the computer on which it is running—isn’t relevant to authentication.

Trusted-host authentication is different. [10] Rather than making you prove your identity to every host that you visit, trusted-host authentication establishes trust relationships between machines. If you are logged in as user andrew on machine A, and you connect by SSH to account bob on machine B using trusted-host authentication, the SSH server on machine B doesn’t check your identity directly. Instead, it checks the identity of host A, making sure that A is a trusted host. It further checks that the connection is coming from a trusted program on A, one installed by the system administrator that won’t lie about andrew’s identity. If the connection passes these two tests, the server takes A’s word you have been authenticated as andrew and proceeds to make an authorization check that andrew@A is allowed to access the account bob@B.

[10] The term “trusted-host” is our own; it refers to the Rhosts, SSH-1 RhostsRSA, and SSH-2 hostbased authentication methods as a related group.

Let’s follow this authentication process step by step:

1. The SSH client requests a connection from the SSH server.
2. The SSH server uses its local naming service to look up a hostname for the source address of the client network connection.

3. The SSH server consults authorization rules in several local files, indicating whether particular hosts are trusted or not. If the server finds a match for the hostname, authentication continues; otherwise it fails.

4. The server verifies that the remote program is a trusted one by following the old Unix convention of privileged ports. Unix-based TCP and UDP stacks reserve the ports numbered 1 through 1023 as privileged, allowing only processes running as root to listen on them or use them on the local side of a connection. The server simply checks that the source port of the connection is in the privileged range. Assuming the client host is secure, only its superuser can arrange for a program to originate such a connection, so the server believes it is talking to a trusted program.

5. If all goes well, authentication succeeds.

This process has been practiced for years by the Berkeley r-commands: `rsh`, `rlogin`, `rcp`, `rexec`, etc. Unfortunately, it is a notoriously weak authentication method within modern networks. IP addresses can be spoofed, naming services can be subverted, and privileged ports aren't so privileged in a world of desktop PCs whose end users commonly have superuser (administrator) privileges. Indeed, some desktop operating systems lack the concept of a user (such as MacOS), while others don't implement the privileged-port convention (Windows), so any user may access any free port.

Nevertheless, trusted-host authentication has advantages. For one, it is simple: you don't have to type passwords or passphrases, or generate, distribute, and maintain keys. It also provides ease of automation. Unattended processes such as `cron` jobs may have difficulty using SSH if they need a key, passphrase, or password coded into a script, placed in a protected file, or stored in memory. This isn't only a potential security risk but also a maintenance nightmare. If the authenticator ever changes, you must hunt down and change these hard coded copies, a situation just begging for things to break mysteriously later on. Trusted-host authentication gets around this problem neatly.

Since trusted-host authentication is a useful idea, SSH1 supports it in two ways. `Rhosts authentication` simply behaves as described in Steps 1-5, just like the Berkeley r-commands. This method is disabled by default, since it is quite insecure, though it's still an improvement over `rsh` since it provides server host authentication, encryption, and integrity. More importantly, though, SSH1 provides a more secure version of the trusted-host method, called `RhostsRSA authentication`, which improves Steps 2 and 4 using the client's host key.

Step 2 is improved by a stronger check on the identity of the client host. Instead of relying on the source IP address and a naming service such as DNS, SSH uses public-key cryptography. Recall that each host on which SSH is installed has an asymmetric "host key" identifying it. The host key authenticates the server to the client while establishing the secure connection. In RhostsRSA authentication, the client's host key authenticates the client host to the server. The client host provides its name and public key, and then must prove it holds the corresponding private key via a challenge-response exchange. The server maintains a list of known hosts and their public keys to determine the client's status as a known, trusted host.

Step 4, checking that the server is talking to a trusted program, is improved again through use of the client's host key. The private key is kept protected so only a program with special privileges (e.g., setuid root) can read it. Therefore, if the client can access its local host key at all—which it must do to complete authentication in Step 2—the client must have those special privileges. Therefore the client was installed by the administrator of the trusted host and can be trusted. SSH1 retains the privileged-port check, which can't be turned off. SSH2 does away with this check entirely since it doesn't add anything.

SSH1 has a `UsePrivilegedPort` configuration keyword, but it tells the client not to use a privileged port in its source socket, which renders the session unusable for rhosts or RhostsRSA.
authentication. The purpose of this feature is to get around firewalls that might block connections coming from privileged ports and requires that some other authentication method be used.

### 3.4.2.3.1 Trusted-host access files

Two pairs of files on the SSH server machine provide access control for trusted-host authentication, in both its weak and strong forms:

- `/etc/hosts.equiv` and `~/.rhosts`
- `/etc/shosts.equiv` and `~/.shosts`

The files in `/etc` have machine-global scope, while those in the target account's home directory are specific to that account. The `hosts.equiv` and `shosts.equiv` files have the same syntax, as do the `.rhosts` and `.shosts` files, and by default they are all checked.

If any of the four access files allows access for a particular connection, it's allowed, even if another of the files forbids it.

The `/etc/hosts.equiv` and `~/.rhosts` files originated with the insecure r-commands. For backward compatibility, SSH can also use these files for making its trusted-host authentication decisions. If using both the r-commands and SSH, however, you might not want the two systems to have the same configuration. Also, because of their poor security, it's common to disable the r-commands, by turning off the servers in your `inetd.conf` files and/or removing the software. In that case, you may not want to have any traditional control files lying around, as a defensive measure in case an attacker managed to get one of these services turned on again.

To separate itself from the r-commands, SSH reads two additional files, `/etc/shosts.equiv` and `~/.shosts`, which have the same syntax and meaning as `/etc/hosts.equiv` and `~/.rhosts`, but are specific to SSH. If you use only the SSH-specific files, you can have SSH trusted-host authentication without leaving any files the r-commands would look at.\(^{[12]}\)

Unfortunately, you can't configure the server to look at one set but not the other. If it looks at `~/.shosts`, then it also considers `~/.rhosts`, and both global files are always considered.

All four files have the same syntax, and SSH interprets them very similarly—but not identically—to the way the r-commands do. Read the following sections carefully to make sure you understand this behavior.

### 3.4.2.3.2 Control file details

Here is the common format of all four trusted-host control files. Each entry is a single line, containing either one or two tokens separated by tabs and/or spaces. Comments begin with `#`, continue to the end of the line, and may be placed anywhere; empty and comment-only lines are allowed.

```text
# example control file entry
[+-][@]hostspec  [+-][@]userspec  # comment
```

The two tokens indicate host(s) and user(s), respectively; the `userspec` may be omitted. If the at-sign (`@`) is present, then the token is interpreted as a netgroup (see Sidebar "Netgroups"), looked up using the `innetgr()` library call, and the resulting list of user or hostnames is substituted. Otherwise, the token is interpreted as a single host or username. Hostnames must be canonical as reported by `gethostbyaddr()` on the server host; other names won't work.
Netgroups

A netgroup defines a list of (host, user, domain) triples. Netgroups are used to define lists of users, machines, or accounts, usually for access-control purposes; for instance, one can usually use a netgroup to specify what hosts are allowed to mount an NFS filesystem (e.g., in the Solaris share command or BSD exportfs).

Different flavors of Unix vary in how they implement netgroups, though you must always be the system administrator to define a netgroup. Possible sources for netgroup definitions include:

- A plain file, e.g., /etc/netgroup
- A database file in various formats, e.g., /etc/netgroup.db
- An information service, such as Sun's YP/NIS

On many modern Unix flavors, the source of netgroup information is configurable with the Network Service Switch facility; see the file /etc/nsswitch.conf. Be aware that in some versions of SunOS and Solaris, netgroups may be defined only in NIS; it doesn't complain if you specify "files" as the source in nsswitch.conf, but it doesn't work either. Recent Linux systems support /etc/netgroup, though C libraries before glibc 2.1 support netgroups only over NIS.

Some typical netgroup definitions might look like this:

```bash
# defines a group consisting of two hosts: hostnames "print1" and "print2", in the (probably NIS) domains one.foo.org and two.foo.com.
print-servers       (print1,,one.foo.com) (print2,,two.foo.com)
# a list of three login servers
login-servers       (login1,,foo.com) (login2,,foo.com)
# Use two existing netgroups to define a list of all hosts, throwing in another.foo.com as well.
all-hosts           print-servers login-servers
                     (another,,foo.com)
# A list of users for some access-control purpose. Mary is allowed from anywhere in the foo.com domain, but Peter only from one host. Alice # is allowed from anywhere at all.
allowed-users       (,mary,foo.com) (login1,peter,foo.com) (,alice,)
```

When deciding membership in a netgroup, the thing being matched is always construed as an appropriate triple. A triple (x, y, z) matches a netgroup N if there exists a triple (a, b, c) in N which matches (x, y, z). In turn, you define that these two triples match if and only if the following conditions are met:

\[ x = a \text{ or } x \text{ is null or } a \text{ is null } \]

and:
This means that a null field in a triple acts as wildcard. By "null," we mean missing; that is, in the triple (, user, domain), the host part is null. This isn't the same as the empty string: ("", user, domain). In this triple, the host part isn't null. It is the empty string, and the triple can match only another whose host part is also the empty string.

When SSH matches a username $U$ against a netgroup, it matches the triple (, $U$); similarly, when matching a hostname $H$, it matches ($H$, ). You might expect it to use (, $U$, $D$) and ($H$, , $D$) where $D$ is the host's domain, but it doesn't.

If either or both tokens are preceded by a minus sign (-), the whole entry is considered negated. It doesn't matter which token has the minus sign; the effect is the same. Let's see some examples before explaining the full rules.

The following hostspec allows anyone from fred.flintstone.gov to log in if the remote and local usernames are the same:

```
# /etc/shosts.equiv
fred.flintstone.gov
```

The following hostspecs allow anyone from any host in the netgroup "trusted-hosts" to log in, if the remote and local usernames are the same, but not from evil.empire.org, even if it is in the trusted-hosts netgroup.

```
# /etc/shosts.equiv
-evil.empire.org
@trusted-hosts
```

This next entry (hostspec and userspec) allows mark@way.too.trusted to log into any local account! Even if a user has -way.too.trusted mark in ~/.shosts, it won't prevent access since the global file is consulted first. You probably never want to do this.

```
# /etc/shosts.equiv
way.too.trusted mark
```

On the other hand, the following entries allow anyone from sister.host.org to connect under the same account name, except mark, who can't access any local account. Remember, however, that a target account can override this restriction by placing sister.host.org mark in ~/.shosts. Note also, as shown earlier, that the negated line must come first; in the other order, it's ineffective.

```
# /etc/shosts.equiv
sister.host.org -mark
sister.host.org
```

This next hostspec allows user wilma on fred.flintstone.gov to log into the local wilma account:

```
# ~wilma/.shosts
fred.flintstone.gov
```
This entry allows user fred on \texttt{fred.flintstone.gov} to log into the local wilma account, but no one else—not even \texttt{wilma@fred.flintstone.gov}:

\begin{verbatim}
# ~wilma/.shosts
fred.flintstone.gov fred
\end{verbatim}

These entries allow both fred and wilma on \texttt{fred.flintstone.gov} to log into the local wilma account:

\begin{verbatim}
# ~wilma/.shosts
fred.flintstone.gov fred
fred.flintstone.gov
\end{verbatim}

Now that we’ve covered some examples, let’s discuss the precise rules. Suppose the client username is \texttt{C}, and the target account of the SSH command is \texttt{T}. Then:

1. A \texttt{hostspec} entry with no \texttt{userspec} permits access from all \texttt{hostspec} hosts when \texttt{T} = \texttt{C}.
2. In a per-account file (~\texttt{.rhosts} or ~\texttt{.shosts}), a \texttt{hostspec userspec} entry permits access to the containing account from \texttt{hostspec} hosts when \texttt{C} is any one of the \texttt{userspec} usernames.
3. In a global file (/\texttt{etc/hosts.equiv} or /\texttt{etc/shosts.equiv}), a \texttt{hostspec userspec} entry permits access to any local target account from any \texttt{hostspec} host, when \texttt{C} is any one of the \texttt{userspec} usernames.
4. For negated entries, replace "permits" with "denies" in the preceding rules.

Note Rule #3 carefully. You never, ever want to open your machine to such a security hole. The only reasonable use for such a rule is if it is negated, thus disallowing access to any local account for a particular remote account. We present some examples shortly.

The files are checked in the following order (a missing file is simply skipped, with no effect on the authorization decision):

1. /\texttt{etc/hosts.equiv}
2. /\texttt{etc/shosts.equiv}
3. ~\texttt{.shosts}
4. ~\texttt{.rhosts}

SSH makes a special exception when the target user is root: it doesn’t check the global files. Access to the root account can be granted only via the root account’s \texttt{.rhosts} and \texttt{.shosts} files. If you block the use of those files with the \texttt{IgnoreRootRhosts} server directive, this effectively prevents access to the root account via trusted-host authentication.

When checking these files, there are two rules to keep in mind. The first rule is: the first accepting line wins. That is, if you have two netgroups:

\begin{verbatim}
set     (one,,) (two,,) (three,,)
subset  (one,,) (two,,)
\end{verbatim}

the following /\texttt{etc/shosts.equiv} file permits access only from host three:

\begin{verbatim}
-@subset
@set
\end{verbatim}

But this next one allows access from all three:

\begin{verbatim}
@set
-@subset
\end{verbatim}
The second line has no effect, because all its hosts have already been accepted by a previous line.

The second rule is: if any file accepts the connection, it's allowed. That is, if /etc/shosts.equiv forbids a connection but the target user's ~/.shosts file accepts it, then it is accepted. Therefore the sysadmin can't rely on the global file to block connections. Similarly, if your per-account file forbids a connection, it can be overridden by a global file that accepts it. Keep these facts carefully in mind when using trusted-host authentication. [13]

[13] By setting the server's IgnoreRhosts keyword to yes, you can cause the server to ignore the per-account files completely and consult the global files exclusively instead. [Section 5.5.1.3]

3.4.2.3.3 Netgroups as wildcards

You may have noticed the rule syntax has no wildcards; this omission is deliberate. The r-commands recognize bare + and - characters as positive and negative wildcards, respectively, and a number of attacks are based on surreptitiously adding a "+" to someone's .rhosts file, immediately allowing anyone to rlogin as that user. So SSH deliberately ignores these wildcards. You'll see messages to that effect in the server's debugging output if it encounters such a wildcard:

Remote: Ignoring wild host/user names in /etc/shosts.equiv

However, there's still a way to get the effect of a wildcard: using the wildcards available in netgroups. An empty netgroup:

empty  # nothing here

matches nothing at all. However, this netgroup:

wild  (,,)

matches everything. In fact, a netgroup containing (,,) anywhere matches everything, regardless of what else is in the netgroup. So this entry:

# ~/.shosts
@wild

allows access from any host at all, [14] as long as the remote and local usernames match. This one:

# ~/.shosts
way.too.trusted @wild

allows any user on way.too.trusted to log into this account, while this entry:

# ~/.shosts
@wild @wild

allows any user access from anywhere.

Given this wildcard behavior, it's important to pay careful attention to netgroup definitions. It's easier to create a wildcard netgroup than you might think. Including the null triple (,,) is the obvious approach. However, remember that the order of elements in a netgroup triple is (host,user,domain). Suppose you define a group "oops" like this:
You intend for this to be a group of usernames, but you've placed the usernames in the host slots, and the username fields are left null. If you use this group as the userspec of a rule, it will act as a wildcard. Thus this entry:

# ~/.shosts
home.flintstones.gov @oops

allows anyone on home.flintstones.gov, not just your three friends, to log into your account. Beware!

### 3.4.2.3.4 Summary

Trusted-host authentication is convenient for users and administrators, because it can set up automatic authentication between hosts based on username correspondence and inter-host trust relationships. This removes the burden of typing passwords or dealing with key management. However, it is heavily dependent on the correct administration and security of the hosts involved; compromising one trusted host can give an attacker automatic access to all accounts on other hosts. Also, the rules for the access control files are complicated, fragile, and easy to get wrong in ways that compromise security. In an environment more concerned with eavesdropping and disclosure than active attacks, it may be acceptable to deploy RhostsRSA (SSH-2 "hostbased") authentication for general user authentication. In a more security-conscious scenario, however, it is probably inappropriate, though it may be acceptable for limited use in special-purpose accounts, such as for unattended batch jobs. [Section 11.1.3]

We don't recommend the use of weak ("Rhosts") trusted-host authentication at all in SSH1 and OpenSSH/1. It is totally insecure.

### 3.4.2.4 Kerberos authentication

SSH1 and OpenSSH provide support for Kerberos-based authentication; SSH2 doesn't yet. [15] [Section 11.4] Table 3-2 summarizes the support features in these products.

[15] At press time, experimental Kerberos support is being integrated into SSH2 2.3.0.

<table>
<thead>
<tr>
<th>Product</th>
<th>Kerberos Version</th>
<th>Tickets</th>
<th>Password Authentication</th>
<th>AFS</th>
<th>Forwarding</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSH1</td>
<td>5</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>OpenSSH</td>
<td>4</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Only with AFS</td>
</tr>
</tbody>
</table>

The following list explains the columns:

**Tickets**

Performs standard Kerberos authentication. The client obtains a ticket for the "host" (v5) or "rcmd" (v4) service on the server and sends that to the SSH server as proof of identity; the server validates it in the standard fashion. Both SSH1 and OpenSSH do Kerberos mutual authentication. This isn't strictly necessary given that SSH has already authenticated the server as part of connection setup, but the extra check can't hurt.

**Password Authentication**
Option to perform server-side password authentication using Kerberos. Instead of checking the password using the operating system's account database, the SSH server instead attempts to obtain Kerberos initial credentials for the target user (a "ticket-granting-ticket" or TGT). If this succeeds, the user is authenticated. Also, the server stores the TGT for the session so that the user has access to it, thus removing the need for an explicit `kinit`.

**AFS**

The Andrew File System (http://www.faqs.org/faqs/afs-faq/), or AFS, uses Kerberos-4 in a specialized way for its authentication. OpenSSH has extra support for obtaining and forwarding AFS credentials. This can be critical in environments using AFS for file sharing. Before it performs authentication, `sshd` must read the target account's home directory, for instance to check `~/.shosts`, or `~/.ssh/authorized_keys`. If the home directory is shared via AFS, then depending on AFS permissions `sshd` might not be able to read it unless it has valid AFS credentials for the owning user. The OpenSSH AFS code provides this, forwarding the source user's Kerberos-4 TGT and AFS ticket to the remote host for use by `sshd`.

**Forwarding**

Kerberos credentials are normally usable only on the machine to which they are issued. The Kerberos-5 protocol allows a user to forward credentials from one machine to another on which he has been authenticated, avoiding the need for repeated `kinit` invocations. SSH1 supports this with the `KerberosTgtPassing` option. Kerberos-4 doesn't do ticket forwarding, so OpenSSH doesn't provide this feature—unless it is using AFS, whose modified Kerberos-4 implementation provides a form of ticket forwarding.

OpenSSH provides Kerberos support only when using the SSH-1 protocol.

### 3.4.2.5 One-time passwords

Password authentication is convenient because it can be used easily from anywhere. If you travel a lot and use other people's computers, passwords might be your best bet for SSH authentication. However, it's precisely in that situation that you're most concerned about someone stealing your password—by monitoring keyboard activity on a hacked computer or by old-fashioned shoulder-surfing. One-time password, or OTP systems, preserve the convenience of password access while mitigating the risk: each login requires a different, unpredictable password. Here are the properties of some OTP systems:

- With the free S/Key software OTP system, you carry a printed list of passwords or calculate the next one needed using a piece of software on your laptop or PDA.
- With the SecurID system from RSA Security, Inc., you carry a small hardware token (credit-card or key-fob size) with an LCD screen, which displays a passcode that changes frequently and is synchronized with the SecurID server, which verifies the passcode.
- The OTP system from Trusted Information Systems, Inc. (TIS) is a variant called challenge-response: the server displays a challenge, which you type into your software or hardware token. The token supplies the corresponding response, which you supply to be authenticated.

SSH1 supports SecurID as a variant behavior of password authentication, and TIS as a separate method with the `TISAuthentication` configuration keyword (as noted earlier, this is actually a
separate authentication type in the SSH-1 protocol). OpenSSH doesn't support TIS but instead reuses the TIS message types in the SSH-1 protocol to implement S/Key. This works because both TIS and S/Key fit the model of a challenge/response exchange.

Using these systems involves obtaining the requisite libraries and header files, compiling SSH with the appropriate configure switches, enabling the right SSH authentication method, and setting up the system according to its instructions. If you are using SecurID or TIS, the requisite libraries and header files should have come with the software or be available from the vendor. S/Key is widely available on the Net, though it has diverged into many versions, and we don't know a canonical site for it. One popular implementation is found in the logdaemon package by Wietse Venema; see http://www.porcupine.org/wietse/. The details of these external packages are mostly outside the scope of SSH proper, so we won't delve into them.

### 3.4.3 Integrity Checking

The SSH-1 protocol uses a weak integrity check: a 32-bit cyclic redundancy check or CRC-32. This sort of check is sufficient for detecting accidental changes to data, but isn't effective against deliberate corruption. In fact, the "insertion attack" of Futoransky and Kargieman specifically targets this weakness in SSH-1. [Section 3.10.5] The use of the CRC-32 integrity check is a serious inherent weakness in SSH-1 that helped prompt the evolution of SSH-2, which uses cryptographically strong integrity checking invulnerable to this attack.

### 3.4.4 Compression

The SSH-1 protocol supports compression of session data using the "deflate" algorithm of the GNU gzip utility (ftp://ftp.gnu.org/pub/gnu/gzip/). Packet data bytes in each direction are compressed separately, each as a single large stream without regard to packet boundaries.

While not typically needed on LAN or fast WAN links, compression can improve speed noticeably over slower links, such as an analog modem line. It is especially beneficial for file transfers, X forwarding, and running curses-style programs in a terminal session, such as text editors. Also, since compression is done before encryption, using compression can reduce delays due to encryption. This may be especially effective with 3DES, which is quite slow.

### 3.5 Inside SSH-2

In this section, we discuss the design and internals of SSH-2, focusing particularly on its differences and improvements as compared to SSH-1. We won't repeat the information common to the two protocols. We also compare the products SSH1 and SSH2, their software implementation differences, and their protocol support. Figure 3-4 summarizes the architecture of SSH-2.

![Figure 3.4. SSH-2 architecture](image-url)
The most important distinction between SSH1 and SSH2 is that they support different, incompatible versions of the SSH protocol: SSH-1.5 and SSH-2.0. [Section 1.5] These products also have important implementation differences, some due to the differing protocols, but many are simply omissions due to SSH2's being a complete rewrite.

### 3.5.1 Protocol Differences (SSH-1 Versus SSH-2)

SSH-1 is monolithic, encompassing multiple functions in a single protocol. SSH-2, on the other hand, has been separated into modules and consists of three protocols working together:

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

Each of these protocols has been specified separately, and a fourth document, SSH Protocol Architecture (SSH-ARCH), describes the overall architecture of the SSH-2 protocol as realized in these three separate specifications.
In a pattern, each service is a name indicating a server to which this pattern applies. SSH recognizes the following service names:

**sshd**

The main SSH server. This can be `sshd`, `sshd1`, `sshd2`, or whatever name you invoke the daemon under (its `argv[0]` value).

**sshdfwd-x11**

The X forwarding port.

**sshdfwd-N**

Forwarded TCP port \( N \) (e.g., forwarded port 2001 is `service sshdfwd-2001`).

The X and port forwarding control features are available only in SSH1 and SSH2; OpenSSH uses only `libwrap` to control access to the main server.

Each client is a pattern that matches a connecting client. It can be:

- An IP address in dotted-quad notation (e.g., 192.168.10.1).
- A hostname (DNS, or whatever naming services the host is using).
- An IP network as network-number/mask (e.g., 192.168.10.0/255.255.255.255; note that the "/n-mask-bits" syntax, 192.168.10.0/24, isn't recognized).
- "ALL", matching any client source address.

**Example 9.1** shows a sample `/etc/hosts.allow` configuration. This setup allows connections to any service from the local host's loopback address, and from all addresses 192.168.10.x. This host is running publicly available servers for SSH1, POP, and IMAP, so we allow connections to these from anywhere, but SSH-2 clients are restricted to sources in another particular range of networks.

**Example 9.1. Sample /etc/hosts.allow File**

```bash
# # network access control for programs invoked by tcpd (see inetd.conf) or # using libwrap. See the manpages hosts_access(5) and hosts_options(5). # allow all connections from my network or localhost (loopback address) # ALL : 192.168.10.0/255.255.255.0 localhost # allow connections to these services from anywhere # ipop3d imapd sshd1 : ALL # allow SSH-2 connections from the class C networks # 192.168.20.0, 192.168.21.0, ..., 192.168.27.0 # sshd2 : 192.168.20.0/255.255.248.0 # allow connections to forwarded port 1234 from host blynken sshdfwd-1234 : blynken.sleepy.net
```

---

293
# restrict X forwarding access to localhost
ss hdfwd-x11 : localhost

# deny everything else
#
ALL : ALL : DENY

We allow connections to the forwarded port 1234 from a particular host, blynken.sleepy.net. Note
that this host doesn't have to be on any of the networks listed so far but can be anywhere at all.
The rules so far say what is allowed, but don't by themselves forbid any connections. So for
example, the forwarding established by the command ssh1 -L1234:localhost:21 remote is
accessible only to the local host, since SSH1 defaults to binding only the loopback address in any
case. But ssh1 -g -L1234:localhost:21 remote is accessible to blynken.sleepy.net as well, as is
either command using ssh2 instead (since ssh2 doesn't affect the localhost restriction and ignores
the -g option). The important difference is that with this use of TCP-wrappers, sshd rejects
connections to the forwarded port, 1234, from any other address.

The sshdfwd-x11 line restricts X-forwarding connections to the local host. This means that if
ssh connects to this host with X forwarding, only local X clients can use the forwarded X
connection. X authentication does this already, but this configuration provides an extra bit of
protection.[8]

[8] SSH2 2.1.0 has a bug that causes an SSH session to freeze after it rejects a forwarded
connection because of a TCP-wrappers rule, at least on some Unix systems. Until this bug is fixed,
don't use TCP-wrappers for protecting forwarded ports (although using it to restrict access to the
main sshd2 server appears to work).

The final line denies any connection that doesn't match the earlier lines, making this a default-to-
closed configuration. If you wanted instead to deny some particular connections but allow all
others, you would use something like this:

ALL : evil.mordor.net : DENY
telnetd : completely.horked.edu : DENY
ALL : ALL : ALLOW

The final line is technically not required, but it's a good idea to make your intentions explicit. If
you don't have the host_options syntax available, you instead have an empty hosts.allow file and
the following lines in hosts.deny:

ALL : evil.mordor.net
telnetd : completely.horked.edu

9.4.2 Notes About TCP-wrappers

Here are a few things to remember when using TCP-wrappers:

- You can't distinguish between ports forwarded by SSH1 and SSH2: the sshdfwd-* rules
  refer to both simultaneously. You can work around this limitation by linking each against
  a different libwrap.a, compiled with different filenames for the allow and deny files, or by
  patching the ssh and sshd executables directly, but then you have to keep track of these
  changes and extra files.
- The big drawback to TCP-wrappers is that it affects all users simultaneously. An
  individual user can't specify custom access rules for himself; there's just the single set of
  global configuration files for the machine. This limits its usefulness on multiuser
  machines.
- If you compile SSH with the --with-libwrap option, it is automatically and always
  turned on; there's no configuration or command-line option to disable the TCP-wrappers
check. Remember that SSH does this check not only for forwarded ports and X
connections but for connections to the main SSH server. As soon as you install a version
of sshd with TCP-wrappers, you must ensure that the TCP-wrappers configuration allows
connections to the server, for instance with the rule sshd1 sshd2 sshd : ALL in
/etc/hosts.allow.

- Using hostnames instead of addresses in the TCP-wrappers rule set involves the usual
  security tradeoff. Names are more convenient, and their use avoids breakage in the future
  if a host address changes. On the other hand, an attacker can potentially subvert the
  naming service and circumvent the access control. If the host machine is configured to
  use only its /etc/hosts file for name lookup, this may be acceptable even in a highly secure
  environment.

- The TCP-wrappers package includes a program called tcpdchk. This program examines
  the wrapper control files and reports inconsistencies that might signal problems. Many
  sites run this periodically as a safety check. Unfortunately, tcpdchk is written only with
  explicit wrapping via inetd.conf in mind. It doesn't have any way of knowing about
  programs that refer to the control files via the libwrap routines, as does sshd. When
  tcpdchk reads control files with SSH rules, it finds uses of the service names sshd1,
  sshdfwd-\, etc., but no corresponding wrapped services in inetd.conf, and it generates a
  warning. Unfortunately, we know of no workaround.

### 9.5 Summary

In this chapter, we discussed SSH port forwarding and X forwarding. Port forwarding is a general
TCP proxying feature that tunnels TCP connections through an SSH session. This is useful for
securing otherwise insecure protocols running on top of TCP or for tunneling TCP connections
through firewalls that would otherwise forbid access. X forwarding is a special case of port
forwarding for X Window System connections, for which SSH has extra support. This makes it
easy to secure X connections with SSH, which is good because X, while popular and useful, is
notoriously insecure. Access control on forwarded ports is normally coarse, but you can achieve
finer control with the TCP-wrappers feature.
Chapter 10. A Recommended Setup

We've just covered a pile of chapters on SSH configuration: is your head spinning yet? With so many choices, you might be wondering which options you should use. How can system administrators secure their systems most effectively with SSH?

When set up properly, SSH works well and invisibly, but sometimes a good setup takes a few tries. In addition, there are some ways to configure the software that are simply wrong. If you're not careful, you can introduce security holes into your system.

In this chapter we present a recommended set of options for compilation, server configuration, key management, and client configuration. We assume:

- You're running SSH on a Unix machine.
- You want a secure system, sometimes at the expense of flexibility. For instance, rather than tell you to maintain your \texttt{.rhosts} files carefully, we recommend disabling Rhosts authentication altogether.

Of course, no single configuration covers all the possibilities; that is, after all, the point of configuration. This is just a sample setup, more on the secure side, to give you a starting point and cover some of the issues involved.

10.1 The Basics

Before you start configuring, make sure you're running an up-to-date SSH version. Some older versions have known security holes that are easily exploited. Always run the latest stable version, and apply updates or patches in a timely manner. (The same goes for your other security software.)

Always keep important SSH-related files and directories protected. The server's host key should be readable only by root. Each user's home directory, SSH configuration directory, and \texttt{.rhosts} and \texttt{.shosts} files should be owned by the user and protected against all others.

Also, remember that SSH doesn't and can't protect against all threats. It can secure your network connections but does nothing against other types of attacks, such as dictionary attacks against your password database. SSH should be an important part, but not the only part, of a robust security policy. [Section 3.11]

10.2 Compile-Time Configuration

In Chapter 4, we covered many compile-time flags for building SSH distributions. Several flags should be carefully set to make your server machine maximally secure:

\texttt{--with-etcdir=... (SSH1, SSH2)}

Make sure your \texttt{etc} directory is on a local disk, not an NFS-mounted partition. If the SSH server reads a file via NFS, the contents are transmitted in the clear across the network, violating security. This is especially true of the host key, which is stored unencrypted in this directory.
Likewise, make sure your SSH executables are installed on a local disk, as they can be spoofed if loaded over NFS.

-enable-suid-ssh (SSH1)
-enable-suid-ssh-signer (SSH2)

Our recommended serverwide configuration disables trusted-host authentication, so there's no need for setuid permissions for ssh1 and ssh-signer2.

-enable-none (SSH1)

You should disable the "none" cipher that permits unencrypted transmissions. An intruder with access to a user account for 10 seconds can add "Ciphers None" to its client configuration file, silently disabling encryption for the user's clients. If you need the none cipher for testing, build a separate server using —with-none and make it executable only by the system administrator.

-enable-rsh (SSH1, OpenSSH)

We don't recommend allowing ssh to fall back to rsh. You can enforce this restriction at compile time using —without-rsh, or at runtime in the serverwide configuration file. The choice is yours.

-enable-libwrap (SSH1, SSH2)
-enable-tcp-wrappers (OpenSSH)

Libwrap affords more precise control over which client machines are allowed to connect to your server. It also makes port and X forwarding more flexible, since otherwise local forwardings are available either only to the local host or from anywhere at all. With GatewayPorts (or ssh -g) and libwrap, you can limit forwarding access to specific hosts. [Section 9.2.1.1]

10.3 Serverwide Configuration

Chapter 5 was a detailed discussion of sshd and how to configure its runtime behavior. Now let's determine which configuration options are most important for security.

10.3.1 Disable Other Means of Access

SSH can provide a secure front door into your system but don't forget to close the back doors. If your system allows access via the infamous r-commands, disable them. This means:

- Remove the file /etc/hosts.equiv, or make it a read-only empty file.
- Disable rshd, rlogind, and rexecd by removing or commenting out their lines in /etc/inetd.conf:

```
# turned off -- don't use!
#shell stream tcp nowait root /usr/sbin/in.rshd
in.rshd
```

297
Make sure you restart `inetd` after doing this, so that the change takes effect (e.g., `skill -HUP inetd`).

- Educate users not to create `.rhosts` files.

You might also consider disabling `telnetd` and other insecure avenues for logging in, permitting logins only via SSH.

### 10.3.2 /etc/sshd_config

We'll now discuss our recommended `sshd_config` settings. We have omitted some keywords that aren't particularly security-related, such as `PrintMotd`, which simply prints a message after login. For any remaining keywords, use your judgment based on your system and needs.

The following files may be located anywhere on the machine's local disk. For security's sake, don't put them on an NFS-mounted partition. If you do, each time the files are accessed by the SSH server, their contents are transmitted in the clear over the network.

- `HostKey /etc/ssh_host_key`
- `PidFile /etc/sshd.pid`
- `RandomSeed /etc/ssh_random_seed`

The following settings control file and directory permissions. The `StrictModes` value requires users to protect their SSH-related files and directories, or else they can't authenticate. The `Umask` value causes any files and directories created by `sshd` to be readable only by their owner (the uid under which `sshd` is running).

- `StrictModes yes`
- `Umask 0077`

The following code represents the server's TCP settings. The `Port` and `ListenAddress` values are standard. We set an idle timeout to reduce the chances that an unattended terminal can be misused by an intruder. Fifteen minutes is short enough to be useful but long enough not to annoy users, though this depends on usage patterns. You may certainly use your judgment and set a different value, but do think about the issue. Also, we enable keepalive messages so connections to clients that have crashed or otherwise become unreachable will terminate rather than hang around and require manual reaping by the sysadmin.

- `Port 22`
- `ListenAddress 0.0.0.0`
- `IdleTimeout 15m`
- `KeepAlive yes`

For logins we allow 30 seconds for a successful authentication, which should be long enough for users and automated processes:

- `LoginGraceTime 30`

The following settings control generation of the server key. We recommend at least 768 bits in your server key and that you regenerate the key at least once an hour (3600 seconds).

- `ServerKeyBits 768`
- `KeyRegenerationInterval 3600`
The following settings control authentication, and we enable only public-key authentication. Password authentication is disabled because passwords can be stolen and used more easily than public keys. This is a fairly harsh restriction, so you might want to leave it enabled depending on your needs. Without password authentication, you have a "chicken and egg" problem: how do users upload their public keys securely the first time? As system administrator, you have to institute a process for this transfer: for example, users can generate keys on a client machine and then request that you install them on the server machine. Rhosts authentication is disabled because it can be spoofed. RhostsRSA authentication is disabled too, because overall it is a medium-security method and this configuration is on the side of higher security.

```
PasswordAuthentication no
RhostsAuthentication no
RhostsRSAAuthentication no
RSASubsystem yes
```

Although we've disabled trusted-host authentication already, we still forbid `sshd` to use `.rhosts` files at all (just in case you reenable trusted-host authentication):

```
IgnoreRhosts yes
IgnoreRootRhosts yes
```

UseLogin is disabled to prevent the unlikely but unwanted use of an alternative login program. (This isn't very useful. An intruder who can install an alternative login program probably can also edit `sshd_config` to change this line.)

```
UseLogin no
```

The following settings limit access to the server, permitting SSH connections only from within the local domain[1] (except for the account fred, which may receive connections from anywhere). If you want to restrict access to particular local accounts or Unix groups, add `AllowUsers` and `AllowGroups` lines (or `DenyUsers` and `DenyGroups`). We have set `SilentDeny` so that any rejections by `DenyHosts` produce no messages for the user. No sense in giving an intruder a clue about what happened, although this can make troubleshooting more difficult.

```
AllowHosts fred@* *.your.domain.com
SilentDeny yes
```

We permit the superuser to connect via SSH but not by password authentication. This is redundant but consistent with turning off `PasswordAuthentication`.

```
PermitRootLogin nopwd
```

For logging error messages, we disable `FascistLogging` because it writes user-specific information to the log, such as the dates and times each person logged in. This information can be valuable to an intruder. We disable `QuietMode`, however, to receive more detailed (but less sensitive) log messages.

```
FascistLogging no
QuietMode no
```

We permit TCP port forwarding and X forwarding so users can secure their other TCP connections:

---
[1] The reliability of this restriction depends on the integrity of DNS. Unfortunately, due to the implementation of `AllowHosts`, restriction by IP address is no more secure. [Section 5.5.2.1]
AllowTcpForwarding yes
X11Forwarding yes

**10.3.3 /etc/ssh2/sshd2_config**

We now move to our recommended `sshd2_config` settings. Again, we omitted some keywords that are not security-related.

As we have mentioned before, make sure all SSH-related files are on local disks, not remotely mounted partitions:

```
HostKeyFile /etc/ssh2/hostkey
PublicHostKeyFile /etc/ssh2/hostkey.pub
RandomSeedFile /etc/ssh2/random_seed
```

For the following settings, consider the pros and cons of storing user files on NFS-mounted filesystems: [Section 10.7]

```
UserConfigDirectory
IdentityFile
AuthorizationFile
```

For this setting, see the discussion for SSH1:

```
StrictModes yes
```

For the first three settings, use the same reasoning as for SSH1. `RequireReverseMapping` is trickier, however. You might think that security would be increased by reverse DNS lookups on incoming connections, but in fact, DNS isn't secure enough to guarantee accurate lookups. Also, due to other issues in your Unix and network environment, reverse DNS mappings might not even work properly. [Section 5.4.3.7]

```
Port 22
ListenAddress 0.0.0.0
KeepAlive yes
RequireReverseMapping no
```

For this setting, see the discussion for SSH1:

```
LoginGraceTime 30
```

In addition, since `sshd2` doesn't have a configuration keyword for the number of bits in the server key, run it with the `-b` option:

```
$ sshd2 -b 1024 ...
```

These settings mirror those for SSH1:

```
AllowedAuthentications publickey
RequiredAuthentications publickey
```

You disable `UserKnownHosts` to prevent users from extending trust to unknown hosts for the purpose of trusted-host authentication. The superuser can still specify trusted hosts in `/etc/ssh2/knownhosts`. 

300
IgnoreRhosts yes
UserKnownHosts no

See the discussion for SSH1 about this setting:

PermitRootLogin nopwd

Use either of the following settings as fits your needs. The notable feature is that they both exclude the none cipher which, as discussed for --without-none, may be a security risk.

Ciphers anycipher
Ciphers anystdcipher

The following settings produce enough logging information to be useful:

QuietMode no
VerboseMode yes

Since SSH-2 is a more secure protocol, we have disabled SSH-1 compatibility mode. However, this may not be an option for practical reasons; if you turn it on, set $Sshd1Path to the pathname of your SSH1 server executable.

Ssh1Compatibility no
#Sshd1Path /usr/local/bin/sshd1   # commented out

10.4 Per-Account Configuration

Users should be instructed not to create .rhosts files. If trusted-host authentication is enabled in the local SSH server, advise users to create .shosts files instead of .rhosts.

For SSH1 and OpenSSH, each key in ~/.ssh/authorized_keys should be restricted by appropriate options. First, use the from option to restrict access to particular keys by particular hosts when appropriate. For example, suppose your authorized_keys file contains a public key for your home PC, myhome.isp.net. No other machine will ever authenticate using this key, so make the relationship explicit:

from="myhome.isp.net" ...key...

Also set idle timeouts for appropriate keys:

from="myhome.isp.net",idle-timeout=5m ...key...

Finally, for each key, consider whether port forwarding, agent forwarding, and tty allocation are ever necessary for incoming connections. If not, disable these features with no-port-forwarding, no-agent-forwarding, and no-pty, respectively:

from="myhome.isp.net",idle-timeout=5m,no-agent-forwarding ...key...

10.5 Key Management
We recommend creating user keys at least 1024 bits long. Protect your key with a good passphrase. Make it lengthy and use a mixture of lowercase, uppercase, numeric, and symbolic characters. Don't use words found in a dictionary.

Empty passphrases should be avoided unless you absolutely need to use one, for example, in an automated batch script. [Section 11.1.2.2]

### 10.6 Client Configuration

Most SSH security pertains to the server, but SSH clients have security-related settings too. Here are a few tips:

- Whenever you leave a computer while SSH clients are running, lock the computer's display with a password-protected screen locker. This is particularly important if you're running an agent that permits an intruder to access your remote accounts without a passphrase.
- In your client configuration file, turn on some safety features as mandatory values:

```bash
# SSH1, OpenSSH
# Put at the top of your configuration file
Host *
    FallbackToRsh no
    UseRsh no
    GatewayPorts no
    StrictHostKeyChecking ask

# SSH2 only
# Put at the bottom of your configuration file
*: GatewayPorts no
    StrictHostKeyChecking ask

F8allbackToRsh and UseRsh prevent the insecure r-commands from invocation by SSH without your knowledge. (These aren't present in SSH2.) The GatewayPorts value forbids remote clients from connecting to locally forwarded ports. Finally, rather than blindly connect, the StrictHostKeyChecking value warns you of any changed host keys and asks what you want to do.
```

### 10.7 Remote Home Directories (NFS, AFS)

We've mentioned NFS several times as a potential security risk for SSH installations. Now we delve into more detail on this topic.

In today's world of ubiquitous networking, it is common for your home directory to be shared among many machines via a network file-sharing protocol, such as SMB for Windows machines or NFS and AFS for Unix. This is convenient, but it does raise some issues with SSH, both technical and security-related.

SSH examines files in the target account's home directory in order to make critical decisions about authentication and authorization. For every form of authentication except password, the various
control files in your home directory (authorized_keys, .shosts, .k5login, etc.) enable SSH access to your account. Two things are therefore important:

- Your home directory needs to be safe from tampering.
- SSH must have access to your home directory.

### 10.7.1 NFS Security Risks

The security of shared home directories is often not very high. Although the NFS protocol has versions and implementations that afford greater security, it is woefully insecure in most installations. Often, it employs no reliable form of authentication whatsoever, but rather uses the same scheme as \texttt{rsh}: the source IP address and DNS identify clients, and a privileged source port is proof of trustworthiness. It then simply believes the uid number encoded in NFS requests and grants access as that user. Breaking into a home directory can be as simple as:

1. Discover the uid, and create an account with that uid on a laptop running Unix.
2. Connect that machine to the network, borrowing the IP address of a trusted host.
3. Issue a \texttt{mount} command, \texttt{su} to the account with the uid, and start rifling through the files.

At this point, an intruder can easily add another public key to authorized_keys, and the account is wide open. The moral is that when designing a system, keep in mind that the security of SSH is no stronger than that of the home directories involved. You need at least to be aware of the trade-off between security and convenience involved here. If you are using an insecure NFS and want to avoid this weakness, you can:

- Use SSH2, which has the \texttt{UserConfigDirectory} option to place the per-user SSH configuration files, normally in \texttt{~/.ssh2}, elsewhere, say in \texttt{/var/ssh/username}. You can still set the permissions so their owners can control them, but they won't be shared via NFS and thus not vulnerable. You can do the same with SSH1 or OpenSSH, but as they lack such a configuration option, you need to edit the source code.
- Turn off hostbased authentication, since the \texttt{~/.shosts} control file is vulnerable, and you can't change its location. Or, if you want to use hostbased authentication, set the \texttt{IgnoreRhosts} option. This causes \texttt{sshd} to ignore \texttt{~/.shosts}, relying instead solely on the systemwide \texttt{/etc/shosts.equiv} file.
- If you are truly paranoid, disable swapping on your Unix machine. Otherwise, sensitive information such as server, host, and user keys, or passwords, may be written to disk as part of the normal operation of the Unix virtual memory system (should the running \texttt{sshd} be swapped out to disk). Someone with root access (and a lot of knowledge and luck) could read the swap partition and tease this information out of the mess there—though it's a difficult feat. Another option is to use an operating system that encrypts swap pages on disk, such as OpenBSD.

### 10.7.2 NFS Access Problems

Another problem that can arise with SSH and NFS is one of access rights. With the public-key or trusted-host methods, if the per-user control files are in the usual place, \texttt{sshd} must read the target account's home directory in order to perform authentication. When that directory is on the same machine as \texttt{sshd}, this isn't a problem. \texttt{sshd} runs as root, and therefore has access to all files. However, if the directory is mounted from elsewhere via NFS, \texttt{sshd} might not have access to the directory. NFS is commonly configured so the special access privileges accorded the root account don't extend to remote filesystems.

Now, this isn't a truly serious restriction. Since one of the root privileges is the ability to create a process with any uid, root can simply "become" the right user, and access the remote directory. SSH1 and SSH2 use this mechanism, but OpenSSH doesn't currently have it. [Section 3.6]
You can work around the problem, but to do so you must make your *authorized_keys* file world-readable; the only way to let root read it remotely is to let everyone read it. This isn't too objectionable. The *authorized_keys* file contains no secrets; though you might prefer not to reveal which keys allow access to your account, thus advertising which keys to steal. However, to grant this access, you must make your home directory and `~/.ssh` world-searchable (that is, permissions at least 711). This doesn't allow other users to steal the contents, but it does allow them to guess at filenames and have those guesses verified. It also means that you must be careful about permissions on your files, since the top-level permissions on your directory don't prevent access by others.

All this may be entirely unacceptable or no problem at all; it depends on your attitude towards your files and the other users on the machines where your home directory is accessible.

### 10.7.3 AFS Access Problems

The Andrew File System, or AFS, is a file-sharing protocol similar in purpose to NFS, but considerably more sophisticated. It uses Kerberos-4 for user authentication and is generally more secure than NFS. The access problem discussed previously comes up for AFS, but it's more work to solve, and this time, OpenSSH is the winner.

Since AFS uses Kerberos, access to remote files is controlled by possession of an appropriate Kerberos ticket. There are no uid-switching games root can play; *sshd* must have an appropriate, valid AFS ticket in order to access your home directory. If you are logged into that machine, of course, you can use Kerberos and AFS commands to get such a ticket. However *sshd* needs it before you've logged in, so there's a bit of a quandary.

This need to transfer credentials from machine to machine isn't unique to SSH, of course, and there is a solution for it: ticket forwarding. It takes some special support, because it's sufficient to just copy the ticket over to the remote host; tickets are issued specifically for particular hosts. Ticket forwarding isn't a feature of Kerberos-4 generally (though it is in Kerberos-5), but AFS has implemented it specifically for Kerberos-4 TGTs and AFS access tokens, and OpenSSH performs this forwarding automatically. To use this feature, you must compile both the SSH client and server *--with-kerberos* and *--with-afs*, and turn on `AFSTokenPassing` on both sides (it is on by default). Then, if you have Kerberos-4 and AFS credentials when you log in via SSH, they are automatically transferred to the SSH server, permitting *sshd* access to your home directory to perform public-key or trusted-host authentication.

If you're not using OpenSSH, you might have trouble using SSH in an AFS environment. Patches for SSH1 are available from various sources on the Internet adding the same AFS forwarding features,[2] though we haven't had the opportunity to test them.


### 10.8 Summary

SSH1, SSH2, and OpenSSH are complex and have many options. It is vitally important to understand all options when installing and running SSH servers and clients, so their behavior will conform to your local security policy.

We have presented our recommended options for a high security setting. Your needs may vary. For instance, you might want the flexibility of other authentication methods that we have forbidden in our configuration.
Chapter 11. Case Studies

In this chapter we'll delve deeply into some advanced topics: complex port forwarding, integration of SSH with other applications, and more. Some interesting features of SSH don't come to the surface unless examined closely, so we hope you get a lot out of these case studies. Roll up your sleeves, dive in, and have fun.

11.1 Unattended SSH: Batch or cron Jobs

SSH isn't only a great interactive tool but also a resource for automation. Batch scripts, cron jobs, and other automated tasks can benefit from the security provided by SSH, but only if implemented properly. The major challenge is authentication: how can a client prove its identity when no human is available to type a password or passphrase? (We'll just write "password" from now on to mean both.) You must carefully select an authentication method, and then equally carefully make it work. Once this infrastructure is established, you must invoke ssh properly to avoid prompting the user. In this case study, we discuss the pros and cons of different authentication methods for operating an SSH client unattended.

Note that any kind of unattended authentication presents a security problem and requires compromise, and SSH is no exception. Without a human present when needed to provide credentials (type a password, provide a thumbprint, etc.), those credentials must be stored persistently somewhere on the host system. Therefore, an attacker who compromises the system badly enough can use those credentials to impersonate the program and gain whatever access it has. Selecting a technique is a matter of understanding the pros and cons of the available methods, and picking your preferred poison. If you can't live with this fact, you shouldn't expect strong security of unattended remote jobs.

11.1.1 Password Authentication

Rule number 1: forget password authentication if you care about the security of your batch jobs. In order to use password authentication, you must embed the password within the batch script or put it in a file which the script reads, etc. Whatever you do, the location of the password will be obvious to anyone reading the script. We don't recommend this technique; the public-key methods coming up are more secure.

11.1.2 Public-Key Authentication

In public-key authentication, a private key is the client's credentials. Therefore the batch job needs access to the key, which must be stored where the job can access it. You have three choices of location for the key, which we discuss separately:

- Store the encrypted key and its passphrase in the filesystem.
- Store a plaintext (unencrypted) private key in the filesystem, so it doesn't require a passphrase.
- Store the key in an agent, which keeps secrets out of the filesystem but requires a human to decrypt the key at system boot time.

11.1.2.1 Storing the passphrase in the filesystem

In this technique, you store an encrypted key and its passphrase in the filesystem so a script can access them. We don't recommend this method, since you can store an unencrypted key in the
filesystem with the same level of security (and considerably less complication). In either case, you rely solely on the filesystem's protections to keep the key secure. This observation is the rationale for the next technique.

### 11.1.2.2 Using a plaintext key

A plaintext or unencrypted key requires no passphrase. To create one, run `ssh-keygen` and simply press the Return key when prompted for a passphrase (or similarly, remove the passphrase from an existing key using `ssh-keygen -p`). You can then supply the key filename on the `ssh` command line using the `-i` option, or in the client configuration file with the `IdentityFile` keyword. [Section 7.4.2]

Usually plaintext keys are undesirable, equivalent to leaving your password in a file in your account. They are never a good idea for interactive logins, since the SSH agent provides the same benefits in a much more secure fashion. But a plaintext key is a viable option for automation, since the unattended aspect forces us to rely on some kind of persistent state in the machine. The filesystem is one possibility.

Given that the situations of a plaintext key, encrypted key with stored passphrase, and stored password are in a sense all equivalent, there are still three reasons to prefer the plaintext key method:

- SSH provides much better control over account use on the server side with public-key authentication than with password; this is critical when setting up batch jobs, as we'll discuss shortly.
- All other things being equal, public-key authentication is more secure than password authentication, since it doesn't expose the authentication secret to theft by a malicious server.
- It is awkward to supply a password to SSH from another program. SSH is designed to take passwords from a user only: it doesn't read them from standard input but directly opens its controlling terminal to interact with the user. If there is no terminal, it fails with an error. In order to make this work from a program, you need to use a pseudo-terminal to interact with SSH (e.g., use a tool like Expect).

Plaintext keys are frightening, though. To steal the key, an attacker needs to override filesystem protections only once, and this doesn't necessarily require any fancy hacking: stealing a single backup tape will do. That's why for most cases, we recommend the next method.

### 11.1.2.3 Using an agent

The `ssh-agent` provides another, somewhat less vulnerable method of key storage for batch jobs. A human invokes an agent and loads the needed keys from passphrase-protected key files, just once. Thereafter, unattended jobs use this long-running agent for authentication.

In this case, the keys are still in plaintext but within the memory space of the running agent rather than in a file on disk. As a matter of practical cracking, it is more difficult to extract a data structure from the address space of a running process than to gain illicit access to a file. Also, this solution avoids the problem of an intruder's walking off with a backup tape containing the plaintext key.

Security can still be compromised by overriding filesystem permissions, though. The agent provides access to its services via a Unix-domain socket, which appears as a node in the filesystem. Anyone who can read and write that socket can instruct the agent to sign authentication requests and thus gain use of the keys. But this compromise isn't quite so devastating since the attacker can't get the keys themselves through the agent socket. She merely gains use of the keys for as long as the agent is running and as long as she can maintain her compromise of the host.
The agent method does have a down side: the system can’t continue unattended after a reboot. When the host comes up again automatically, the batch jobs won’t have their keys until someone shows up to restart the agent and provide the passphrases to load the keys. This is just a cost of the improved security, and you have a pager, right?

Another bit of complication with the agent method is that you must arrange for the batch jobs to find the agent. SSH clients locate an agent via an environment variable pointing to the agent socket, such as SSH_AUTH.SOCK for the SSH1 and OpenSSH agents. [Section 6.3.2.1] When you start the agent for batch jobs, you need to record its output where the jobs can find it. For instance, if the job is a shell script, you can store the environment values in a file:

```
$ ssh-agent | head -2 > ~/.agent-info
$ cat ~/.agent-info
setenv SSH_AUTH_SOCK /tmp/ssh-res/ssh-12327-agent;
setenv SSH_AGENT_PID 12328;
```

You can add keys to the agent (assuming C shell syntax here):

```
$ source ~/.agent-info
$ ssh-add batch-key
Need passphrase for batch-key (batch job SSH key).
Enter passphrase: **************
```

then instrument any scripts to set the same values for the environment variables:

```
#!/bin/csh
# Source the agent-info file to get access to our ssh-agent.
set agent = ~/.agent-info
if (-r $agent) then
  source $agent
else
  echo "Can't find or read agent file; exiting."
  exit 1
endif
# Now use SSH for something...
ssh -q -o 'BatchMode yes' user@remote-server my-job-command
```

You also need to ensure that the batch jobs (and nobody else!) can read and write the socket. If there’s only one uid using the agent, the simplest thing to do is start the agent under that uid (e.g., as root, do `su <batch_account> ssh-agent ...`). If multiple uids are using the agent, you must adjust the permissions on the socket and its containing directory so that these uids can all access it, perhaps using group permissions.

---

Some operating systems behave oddly with respect to permissions on Unix-domain sockets. Some versions of Solaris, for example, completely ignore the modes on a socket, allowing any process at all full access to it. To protect a socket in such situations, set the containing directory to forbid access. For example, if the containing directory is mode 700, only the directory owner may access the socket. (This assumes there’s no other shortcut to the socket located elsewhere, such as a hard link.)

Using an agent for automation is more complicated and restrictive than using a plaintext key; however, it is more resistant to attack and doesn’t leave the key on disk and tape where it can be stolen. Considering that the agent is still vulnerable to being misused via the filesystem, and that it is intended to run indefinitely, the advantages of this method are debatable. Still, we recommend...
the agent method as the most secure and flexible strategy for automated SSH usage in a security-conscious environment.

11.1.3 Trusted-Host Authentication

If security concerns are relatively light, consider trusted-host authentication for batch jobs. In this case, the "credentials" are the operating system's notion of a process's uid: the identity under which a process is running, which determines what rights it has over protected objects. An attacker need only manage to get control of a process running under your uid, to impersonate you to a remote SSH server. If he breaks root on the client, this is particularly simple, since root may create processes under any uid. The real crux, though, is the client host key: if the attacker gets that, he can sign bogus authentication requests presenting himself as any user at all, and sshd will believe them.

Trusted-host authentication is in many ways the least secure SSH authentication method. [Section 3.4.2.3] It leaves systems vulnerable to transitive compromise: if an attacker gains access to an account on host H, she immediately has access to the same account on all machines that trust H, with no further effort. Also, trusted-host configuration is limited, fragile, and easy to get wrong. Public-key authentication affords both greater security and flexibility, particularly since you can restrict the commands that may be invoked and the client hosts that may connect, using its forced commands and other options in the authorization file.

11.1.4 Kerberos

Kerberos-5 [Section 11.4] contains support for long-running jobs in the form of renewable tickets. While there's no explicit support for these in SSH, a batch job can be designed to use them. As with agent usage, a human performs an initial kinit to get a TGT for the batch account, using the -r switch to request a renewable ticket. Periodically, the batch job uses kinit -R to renew the TGT before it expires. This can be repeated up to the maximum renewable lifetime of the ticket, typically a few days.

Like trusted-host authentication, however, SSH Kerberos support lacks the close authorization controls provided by the public-key options. Even in an installation using Kerberos for user authentication, it's probably best to use some form of public-key authentication for unattended jobs instead. For more information on renewable tickets, see the Kerberos-5 documentation.

11.1.5 General Precautions for Batch Jobs

Regardless of the method you choose, some extra precautions will help secure your environment.

11.1.5.1 Least-privilege accounts

The account under which the automated job runs should have only those privileges needed to run the job, and no more. Don't run every batch job as root just because it's convenient. Arrange your filesystem and other protections so the job can run as a less privileged user. Remember that unattended remote jobs increase the risk of account compromise, so take the extra trouble to avoid the root account whenever possible.

11.1.5.2 Separate, locked-down automation accounts

Create accounts that are used solely for automation. Try not to run system batch jobs in a user account, since you might not be able to reduce its privileges to the smallest set necessary to support the job. In many cases, an automation account doesn't even need to admit interactive logins. If jobs running under its uid are created directly by the batch job manager (e.g., cron), the account doesn't need a password and should be locked.
11.1.5.3 Restricted-use keys

As much as possible, restrict the target account to perform only the work needed for the job. With public-key authentication, automated jobs should use keys that aren't shared by interactive logins. Imagine that someday you might need to eliminate the key for security reasons, and you don't want to affect other users or jobs by this change. For maximum control, use a separate key for each automated task. Additionally, place all possible restrictions on the key by setting options in the authorization file. [Section 8.2] The command option restricts the key to running only the needed remote command, and the from option restricts usage to appropriate client hosts. Consider always adding the following options as well, if they don't interfere with the job:

no-port-forwarding,no-X11-forwarding,no-agent-forwarding,no-pty

These make it harder to misuse the key should it be stolen.

If you're using trusted-host authentication, these restrictions aren't available. In this case, it's best to use a special shell for the account, which limits the commands that may be executed. Since sshd uses the target account's shell to run any commands on the user's behalf, this is an effective restriction. One standard tool is the Unix "restricted shell." Confusingly, the restricted shell is usually named "rsh", but has nothing to do with the Berkeley r-command for opening a remote shell, rsh.

11.1.5.4 Useful ssh options

When running SSH commands in a batch job, it's a good idea to use these options:

ssh -q -o 'BatchMode yes'

The -q option is for quiet mode, preventing SSH from printing a variety of warnings. This is sometimes necessary if you're using SSH as a pipe from one program to another. Otherwise, the SSH warnings may be interpreted as remote program output and confuse the local program. [Section 7.4.15]

The BatchMode keyword tells SSH not to prompt the user, who in this case doesn't exist. This makes error reporting more straightforward, eliminating some confusing SSH messages about failing to access a tty. [Section 7.4.5.4]

11.1.6 Recommendations

Our recommended method for best security with unattended SSH operation is public-key authentication with keys stored in an agent. If that isn't feasible, trusted-host or plaintext-key authentication may be used instead; your local security concerns and needs will determine which is preferable, using the foregoing discussion as a guideline.

To the extent possible, use separate accounts and keys for each job. By doing so, you limit the damage caused by compromising any one account, or stealing any one key. But of course, there is a complexity trade-off here; if you have a hundred batch jobs, separate accounts or keys for each one may be too much to deal with. In that case, partition the jobs into categories according to the privileges they need, and use a separate account and/or key for each category of job.

You can ease the burden of multiple keys by applying a little automation to the business of loading them. The keys can all be stored under the same passphrase: a script prompts for the passphrase, then runs ssh-add multiple times to add the various keys. Or they have different passphrases, and the human inserts a diskette containing the passphrases when loading them. Perhaps the passphrase list itself is encrypted under a single password provided by the human. For
that matter, the keys themselves can be kept on the key diskette and not stored on the filesystem at all: whatever fits your needs and paranoia level.

### 11.2 FTP Forwarding

One of the most frequently asked questions about SSH is, "How can I use port forwarding to secure FTP?" Unfortunately, the short answer is that you usually can't, at least not completely. Port forwarding can protect your account password, but usually not the files being transferred. Still, protecting your password is a big win, since the most egregious problem with FTP is that it reveals your password to network snoopers.\[1\]

\[1\] At least in its usual form. Some FTP implementations support more secure authentication methods, such as Kerberos. There are even protocol extensions that allow for encryption and cryptographic integrity checking of data connections. These techniques aren't widely implemented, however, and plaintext passwords with unprotected data connections are still the norm for FTP servers on the Internet.

---

**Van Dyke's SecureFX (http://www.vandyke.com/)**

Van Dyke Technologies, Inc. has a very useful Windows product, specifically designed to forward FTP over SSH, data connections and all. It is a specialized combination of SSH-2 and FTP clients. It connects to a server host via SSH-2, then connects to the FTP server (also running on that host) via a "tcpip-direct" channel in the SSH-2 session. This is the same mechanism used for local port forwarding in regular SSH-2 clients, but since this is a specially written application, it can talk to the server directly rather than go through a loopback connection to a locally forwarded TCP port.

SecureFX acts as a GUI FTP client. Whenever it needs an FTP data connection, it dynamically creates whatever channels or remote forwardings are necessary for the data ports (more outbound tcpip-direct channels for active FTP, or regular remote forwardings for passive mode). It works very smoothly and we recommend the product.

Note that at press time SecureFX runs only on Windows (98, 95, NT 4.0, and 2000) and requires an SSH-2 server; it doesn't speak SSH-1.

This section explains in detail what you can and can't do with FTP and SSH, and why. Some difficulties are due to limitations of FTP, not only when interacting with SSH, but also in the presence of firewalls and network address translation (NAT). We will discuss each of these situations, since firewalls and NAT are common nowadays, and their presence might be the reason you're trying to forward FTP securely. If you are a system administrator responsible for both SSH and these networking components, we will try to guide you to a general understanding that will help you design and troubleshoot entire systems.

Depending on your network environment, different problems may arise when combining SSH with FTP. Since we can't cover every possible environment, we describe each problem in isolation, illustrating its symptoms and recommending solutions. If you have multiple problems occurring simultaneously, the software behavior you observe might not match the examples we've given. We recommend reading the entire case study once (at least cursorily) before experimenting with your system, so you will have an idea of the problems you might encounter. Afterward, go ahead and try the examples at your computer.

### 11.2.1 The FTP Protocol
To understand the problems between FTP and SSH, you need to understand a bit about the FTP protocol. Most TCP services involve a single connection from client to server on a known, server-side port. FTP, however, involves multiple connections in both directions, mostly to unpredictable port numbers:

- A single control connection for carrying commands from the client and responses from the server. It connects on TCP port 21 and persists for the entire FTP session.
- A number of data connections for transferring files and other data, such as directory listings. For each file transfer, a new data connection is opened and closed, and each one may be on a different port. These data connections may come from the client or the server.

Let's run a typical FTP client and view the control connection. We'll use debug mode (ftp -d) to make visible the FTP protocol commands the client sends on the control connection, since they aren't normally displayed. Debug mode prints these commands preceded by "--->", for example:

--->

You'll also see responses from the server, which the client prints by default. These are preceded by a numerical code:

230 User res logged in.

Here's a session in which the user res connects to an FTP server, logs in, and attempts to change directory twice, once successfully and once not:

$ ftp -d aaor.lionaka.net
Connected to aaor.lionaka.net.
220 aaor.lionaka.net FTP server (SunOS 5.7) ready.
--->
215 UNIX Type: L8 Version: SUNOS
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> user res
--->
230 User res logged in.
ftp> cd rep
--->
250 CWD command successful.
ftp> cd utopia
--->
550 utopia: No such file or directory.
ftp> quit
--->
221 Goodbye.

The control connection can be secured by standard port forwarding because it is on a known port (21). Section 9.2 In contrast, the destination port numbers for data connections are generally not known in advance, so setting up SSH forwarding for these connections is far more difficult. There's a second standard port number associated with FTP, the ftp-data port (20). But this is only the source port for data connections coming from the server; nothing ever listens on it.

Surprisingly, the data connections generally go in the opposite direction from the control one; that is, the server makes a TCP connection back to the client in order to transfer data. The ports on which these connections occur can be negotiated dynamically by the FTP client and server, and
doing so involves sending explicit IP address information inside the FTP protocol. These features of usual FTP operation can cause difficulties when forwarding SSH connections and in other scenarios involving firewalls or NAT.

An alternative FTP mode, called passive mode, addresses one of these problems: it reverses the sense of the data connections, so that they go from the client to the server. Passive mode is a matter of FTP client behavior, and so is determined by a client setting. The behavior of setting up data connections from the server to the client, which we will call active-mode FTP, is traditionally the default in FTP clients, although that's changing. With a command-line client, the passive command switches to passive mode. The internal command that the client sends the server to tell it to enter passive mode is `PASV`. We discuss specific problems and how passive mode solves them, in upcoming sections. Figure 11-1 summarizes the workings of passive and active FTP.

**Figure 11.1. Basic FTP operation: control connection and active- versus passive-mode transfers**

![Diagram of FTP connections](image)

11.2.2 Forwarding the Control Connection

Since the FTP control connection is just a single, persistent TCP connection to a well-known port, you can forward it through SSH. As usual, the FTP server machine must be running an SSH server, and you must have an account on it that you may access via SSH (see Figure 11-2).
Suppose you are logged into the machine `client` and want to connect securely to an FTP server on the machine `server`. To forward the FTP control connection, run a port-forwarding command on `client`:

\[ \text{ssh -L2001:server:21 server} \]

Then, to use the forwarded port:

```
client% ftp localhost 2001
Connected to localhost
220 server FTP server (SunOS 5.7) ready.
Password:
230 User res logged in.
ftp> passive
Passive mode on.
ftp> ls
... and so on
```

There are two important things to notice about the commands we just recommended. We will discuss each.

- The target of the forwarding is `server`, not `localhost`.
- The client uses passive mode.

### 11.2.2.1 Choosing the forwarding target

We chose `server` as the target of our forwarding, not `localhost` (i.e., we didn't use `-L2001:localhost:21`). This is contrary to our previous advice, which was to use `localhost` where possible as the forwarding target. [Section 9.2.8](#) Well, that technique isn't advisable here. Here's what can happen if you do:

```
client% ftp localhost 2001
Connected to localhost
220 client FTP server (SunOS 5.7) ready.
331 Password required for res.
Password:
230 User res logged in.
ftp> ls
... and so on
```
Can't build data connection: Cannot assign requested address.

ftp>

The problem is a bit obscure but can be revealed by an execution trace of the FTP server as it responds to the \texttt{ls} command. The following output was produced by the Linux \texttt{strace} command:\footnote{If you're on a Solaris 2 (SunOS 5) system, the corresponding operating system-supplied program is called \texttt{truss}. There is also an \texttt{strace} program with Solaris, but it is completely unrelated. Solaris 1 (SunOS 4 and earlier) has a \texttt{trace} command, and BSD has \texttt{ktrace}.}

\begin{verbatim}
so_socket(2, 2, 0, "", 1)    = 5
bind(5, 0x0002D614, 16, 3)    = 0
  AF_INET  name = 127.0.0.1  port = 20
connect(5, 0x0002D5F4, 16, 1)  Err#126 EADDRNOTAVAIL
  AF_INET  name = 192.168.10.1  port = 2845
write(1, " 4 2 5  C a n ' t   b u..", 67) = 67
\end{verbatim}

The FTP server is trying to make a TCP connection to the correct client address but from the wrong socket: the \texttt{ftp-data} port on its loopback address, 127.0.0.1. The loopback interface can talk only to other loopback addresses on the same machine. TCP knows this and responds with the error "address not available" (EADDRNOTAVAIL). The FTP server is being careful to originate the data connection from the same address to which the client made the control connection. Here, the control connection has been forwarded through SSH; so to the FTP server, it appears to come from the local host. And because we used the loopback address as the forwarding target, the source address of that leg of the forwarded path (from \texttt{sshd} to \texttt{ftpd}) is also the loopback address. To eliminate the problem, use the server's nonloopback IP address as the target; this causes the FTP server to originate data connections from that address.

You might try to solve this problem using passive mode, since then the server wouldn't originate any connections. But if you try:

\begin{verbatim}
ftp> passive
Passive mode on.
ftp> ls
227 Entering Passive Mode (127,0,0,1,128,133)
ftp: connect: Connection refused
ftp>
\end{verbatim}

In this case, the failure is a slightly different manifestation of the same problem. This time, the server listens for an incoming data connection from the client, but again, it thinks the client is local so it listens on its loopback address. It sends this socket (address 127.0.0.1, port 32901) to the client, and the client tries to connect to it. But this causes the client to try to connect to port 32901 on the client host, not the server! Nothing is listening there, of course, so the connection is refused.

\subsection*{11.2.2.2 Using passive mode}

Note that we had to put the client into passive mode. You will see later that passive mode is beneficial for FTP in general, because it avoids some common firewall and NAT problems. Here, however, it's used because of a specific FTP/SSH problem; if you didn't, here's what happens:

\begin{verbatim}
$ ftp -d localhost 2001
Connected to localhost.
220 server FTP server (SunOS 5.7) ready.
--- USER res
331 Password required for res.
Password:
\end{verbatim}
This is a mirror image of the problem we saw when localhost was the forwarding target, but this time it happens on the client side. The client supplies a socket for the server to connect to, and since it thinks the server is on the local host, that socket is on the loopback address. This causes the server to try connecting to its local host instead of the client machine.

Passive mode can't always be used: the FTP client or server might not support it, or server-side firewall/NAT considerations may prevent it (you'll see an example of that shortly). If so, you can use the GatewayPorts feature of SSH and solve this problem as we did the previous one: use the host's real IP address instead of the loopback. To wit:

```
client% ssh -g -L2001:server:21 server
```

Then connect to the client machine by name, rather than to localhost:

```
client% ftp client 2001
```

This connects to the SSH proxy on the client's nonloopback address, causing the FTP client to listen on that address for data connections. The -g option has security implications, however. [Section 9.2.1.1]

Of course, as we mentioned earlier, it's often the case that active-mode FTP isn't usable. It's perfectly possible that your local firewall/NAT setup requires passive mode, but you can't use it. In that case, you're just out of luck. Put your data on a diskette and contribute to the local bicycle-courier economy.

The various problems we have described, while common, depend on your particular Unix flavor and FTP implementation. For example, some FTP servers fail even before connecting to a loopback socket; they see the client's PORT command and reject it, printing "illegal PORT command". If you understand the reasons for the various failure modes, however, you will learn to recognize them in different guises.

### 11.2.2.3 The "PASV port theft" problem

Trying to use FTP with SSH can be sort of like playing a computer dungeon game: you find yourself in a twisty maze of TCP connections, all of which look alike and none of which seem to go where you want. Even if you follow all of our advice so far, and understand and avoid the pitfalls we've mentioned, the connection might still fail:

```
ftp> passive
Passive mode on.
ftp> ls
connecting to 192.168.10.1:6670
Connected to 192.168.10.1 port 6670
425 Possible PASV port theft, cannot open data connection.
! Retrieve of folder listing failed
```
Assuming you don't decide to give up entirely and move into a less irritating career, you may want to know, "What now?" The problem here is a security feature of the FTP server, specifically the popular wu-ftpd from Washington University. (See http://www.wu-ftpd.org/. This feature might be implemented in other FTP servers, but we haven't seen it.) The server accepts an incoming data connection from the client, then notices that its source address isn't the same as that of the control connection (which was forwarded through SSH and thus comes from the server host). It concludes that an attack is in progress! The FTP server believes someone has been monitoring your FTP control connection, seen the server response to the PASV command containing the listening socket, and jumped in to connect to it before the legitimate client can do so. So the server drops the connection and reports the suspected "port theft" (see Figure 11-3).

There's no way around this problem but to stop the server from performing this check. It's a problematic feature to begin with, since it prevents not only attacks but also legitimate FTP operations. For example, passive-mode operation was originally intended to allow an FTP client to effect a file transfer between two remote servers directly, rather than first fetching the file to the client and then sending it to the second server. This isn't a common practice, but it is part of the protocol design, and the "port theft" check of wu-ftpd prevents its use. You can turn it off by recompiling wu-ftpd without FIGHT_PASV_PORT_RACE (use configure —disable-pasvip). You can also leave the check on but allow certain accounts to use alternate IP addresses for data connections, with the pasv-allow and port-allow configuration statements. See the ftpaccess manpage for details. Note that these features are relatively recent additions to wu-ftpd and aren't in earlier versions.

11.2.3 FTP, Firewalls, and Passive Mode

Recall that in active mode, the FTP data connections go in the opposite direction than you might expect—from the server back to the client. This usual mode of operation (shown in Figure 11-4) often develops problems in the presence of a firewall. Suppose the client is behind a firewall that allows all outbound connections but restricts inbound ones. Then the client can establish a control connection to log in and issue commands, but data-transfer commands such as ls, get, and put will fail, because the firewall blocks the data connections coming back to the client machine. Simple packet-filtering firewalls can't be configured to allow these connections, because they appear as separate TCP destinations to random ports, with no obvious relation to the established FTP control connection.[4] The failure might happen quickly with the message "connection refused," or the connection might hang for a while and eventually fail. This depends on whether the firewall explicitly rejects the connection attempt with an ICMP or TCP RST message, or just silently drops the packets. Note that this problem can occur whether or not SSH is forwarding the control connection.

[4] More sophisticated firewalls can take care of this problem. These products are a cross between an application-level proxy and a packet filter and are often called "transparent proxies" or "stateful
packet filters." Such a firewall understands the FTP protocol and watches for FTP control connections. When it sees a PORT command issued by an FTP client, it dynamically opens a temporary hole in the firewall, allowing the specified FTP data connection back through. This hole disappears automatically after a short time and can only be between the socket given in the PORT command and the server's ftp-data socket. These products often also do NAT and can transparently deal with the FTP/NAT problems we describe next.

Figure 11.4. FTP client behind a firewall

Passive mode usually solves this problem, reversing the direction of data connections so they go from the client to the server. Unfortunately, not all FTP client or servers implement passive-mode transfers. Command-line FTP clients generally use the `passive` command to toggle passive-mode transfers on and off; if it doesn't recognize that command, it probably doesn't do passive mode. If the client supports passive mode but the server doesn't, you may see a message like "PASV: command not understood" from the server. PASV is the FTP protocol command that instructs the server to listen for data connections. Finally, even if passive mode solves the firewall problem, it doesn't help with SSH forwarding, since the ports in question are still dynamically chosen.

Here is an example of the firewall problem, blocking the return data connections:

```bash
$ ftp lasciate.ogni.speranza.org
Connected to lasciate.ogni.speranza.org
220 ProFTPD 1.2.0pre6 Server (Lasciate FTP Server)
[lasciate.ogni.speranza.org] 331 Password required for slade.
Password: 
230 User slade logged in.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> ls
200 PORT command successful.
[...long wait here...]
425 Can't build data connection: Connection timed out
```

Passive mode comes to the rescue:

```bash
ftp> passive
Passive mode on.
ftp> ls
227 Entering Passive Mode (10,25,15,1,12,65)
150 Opening ASCII mode data connection for file list
drwxr-x-x 21 slade  web          2048 May  8 23:29 .
drwxr-xr-x 111 root wheel       10240 Apr 26 00:09 ..
-rw-------   1 slade  other         106 May  8 15:22 .cshrc
```
Now, in discussing the problem of using FTP through a firewall, we didn't mention SSH at all; it is a problem inherent in the FTP protocol and firewalls. However, even when forwarding the FTP control connection through SSH, this problem still applies, since the difficulty is with the data connection, not the control, and those don't go through SSH. So this is yet another reason why you will normally want to use passive mode with FTP and SSH.

11.2.4 FTP and Network Address Translation (NAT)

Passive-mode transfers can also work around another common problem with FTP: its difficulties with network address translation, or NAT. NAT is the practice of connecting two networks by a gateway that rewrites the source and destination addresses of packets as they pass through. One benefit is that you may connect a network to the Internet or change ISPs without having to renumber the network (that is, change all your IP addresses). It also allows sharing a limited number of routable Internet addresses among a larger number of machines on a network using private addresses not routed on the Internet. This flavor of NAT is often called masquerading.

Suppose your FTP client is on a machine with a private address usable only on your local network, and you connect to the Internet through a NAT gateway. The client can establish a control connection to an external FTP server. However, there will be a problem if the client attempts the usual reverse-direction data connections. The client, ignorant of the NAT gateway, tells the server (via a PORT command) to connect to a socket containing the client's private address. Since that address isn't usable on the remote side, the server generally responds "no route to host" and the connection will fail. Figure 11.5 illustrates this situation. Passive mode gets around this problem as well, since the server never has to connect back to the client and so the client's address is irrelevant.

[5] It could be worse, too. The server could also use private addressing, and if you're unlucky, the client's private address might coincidentally match a completely different machine on the server side. It's unlikely, though, that a server-side machine would happen to listen on the random port picked by your FTP client, so this would probably just generate a "connection refused" error.

Figure 11.5. Client-side NAT prevents active-mode FTP transfers

![Diagram](image)

So far, we've listed three situations requiring passive-mode FTP: control connection forwarding, client inside a firewall, and client behind NAT. Given these potential problems with active-mode FTP, and that there's no down side to passive mode we know of, we recommend always using passive mode FTP if you can.

11.2.4.1 Server-side NAT issues
The NAT problem we just discussed was a client-side issue. There is a more difficult problem that can occur if the FTP server is behind a NAT gateway, and you’re forwarding the FTP control connection through SSH.

First, let’s understand the basic problem without SSH in the picture. If the server is behind a NAT gateway, then you have the mirror-image problem to the one discussed earlier. Before, active-mode transfers didn’t work because the client supplied its internal, non-NAT’d address to the server in the `PORT` command, and this address wasn’t reachable. In the new situation, passive-mode transfers don’t work because the server supplies its internal-only address to the client in the `PASV` command response, and that address is unreachable to the client (see Figure 11-6).

![Figure 11.6. Server-side NAT prevents passive-mode FTP transfers](image)

The earlier answer was to use passive mode; here the simplest answer is the reverse: use active mode. Unfortunately, this isn’t very helpful. If the server is intended for general Net access, it should be made useful to the largest number of people. Since client-side NAT and firewall setups requiring passive-mode FTP are common, it won’t do to use a server-side NAT configuration that requires active mode instead; this makes access impossible. One approach is to use an FTP server with special features designed to address this very problem. The `wu-ftp` server we touched on earlier has such a feature. Quoting from the `ftpaccess` (5) manpage:

```
passive address <externalip> <cidr>
```

Allows control of the address reported in response to a PASV command. When any control connection matching the <cidr> requests a passive data connection (PASV), the <externalip> address is reported. NOTE: this does not change the address the daemon actually listens on, only the address reported to the client. This feature allows the daemon to operate correctly behind IP-renumbering firewalls.

For example:

```
passive address 10.0.1.15 10.0.0.0/8
passive address 192.168.1.5 0.0.0.0/0
```

Clients connecting from the class-A network 10 will be told the passive connection is listening on IP-address 10.0.1.15 while all others will be told the connection is listening on 192.168.1.5

Multiple passive addresses may be specified to handle complex, or multi-gatewayed, networks.

This handles the problem quite neatly, unless you happen to be forwarding the FTP control connection through SSH. Site administrators arrange for FTP control connections originating from outside the server’s private network to have external addresses reported in the PASV responses. But the forwarded control connection appears to come from the server host itself, rather than the
outside network. Control connections coming from inside the private network should get the internal address, not the external one. The only way this will work is if the FTP server is configured to provide the external address to connections coming from itself as well as from the outside. This is actually quite workable, as there's little need in practice to transmit files by FTP from a machine back to itself. You can use this technique to allow control-connection forwarding in the presence of server-side NAT or suggest it to the site administrators if you have this problem.

Another way of addressing the server-side NAT problem is to use an intelligent NAT gateway of the type mentioned earlier. Such a gateway automatically rewrites the FTP control traffic in transit to account for address translation. This is an attractive solution in some respects, because it is automatic and transparent; there is less custom work in setting up the servers behind the gateway, and fewer dependencies between the server and network configurations. As it happens, though, this solution is actually worse for our purposes than the server-level one. This technique relies on the gateway's ability to recognize and alter the FTP control connection as it passes through. But such manipulation is exactly what SSH is designed to prevent! If the control connection is forwarded through SSH, the gateway doesn't know there is a control connection, because it's embedded as a channel inside the SSH session. The control connection isn't a separate TCP connection of its own; it's on the SSH port rather than the FTP port. The gateway can't read it because it's encrypted, and the gateway can't modify it even if the gateway can read it, because SSH provides integrity protection. If you're in this situation—the client must use passive-mode FTP, and the server is behind a NAT gateway doing FTP control traffic rewriting—you must convince the server administrator to use a server-level technique in addition to the gateway, specifically to allow forwarding. Otherwise, it's not going to happen, and we see trucks filled with tapes in your future, or perhaps HTTP over SSL with PUT commands.

We have now concluded our discussion of forwarding the control connection of FTP, securing your login name, password, and FTP commands. If that's all you want to do, you are done with this case study. We're going to continue, however, and delve into the murky depths of data connections. You'll need a technical background for this material as we cover minute details and little-known modes of FTP. (You might even wonder if we've accidentally inserted a portion of an FTP book into the SSH book.) Forward, brave reader!

11.2.5 All About Data Connections

Ask most SSH users about forwarding the FTP data connection, and they'll respond, "Sorry, it's not possible." Well, it is possible. The method we've discovered is obscure, inconvenient, and not usually worth the effort, but it works. Before we can explain it, we must first discuss the three major ways that FTP accomplishes file transfers between client and server:

- The usual method
- Passive-mode transfers
- Transfers using the default data ports

We'll just touch briefly on the first two, since we've already discussed them; we'll just amplify with a bit more detail. Then we'll discuss the third mode, which is the least known and the one you need if you really, really want to forward your FTP data connections.

11.2.5.1 The usual method of file transfer

Most FTP clients attempt data transfers in the following way. After establishing the control connection and authenticating, the user issues a command to transfer a file. Suppose the command is `get fichier.txt`, which asks to transfer the file `fichier.txt` from the server to the client. In response to this command, the client selects a free local TCP socket, call it C, and starts listening on it. It then issues a `PORT` command to the FTP server, specifying the socket C. After the server acknowledges this, the client issues the command `RETR fichier.txt`, which tells the server to connect to the previously given socket (C) and send the contents of that file over the new data
connection. The client accepts the connection to C, reads the data, and writes it into a local file also called fichier.txt. When done, the data connection is closed. Here is a transcript of such a session:

$ ftp -d aaor.lionaka.net 
Connected to aaor.lionaka.net.
220 aaor.lionaka.net FTP server (SunOS 5.7) ready.
--> USER res 
331 Password required for res. 
Password: 
--> PASS XXXX 
230 User res logged in. 
--> SYST 
215 UNIX Type: L8 Version: SUNOS 
Remote system type is UNIX.
Using binary mode to transfer files. 
ftp> get fichier.txt
local: fichier.txt remote: fichier.txt
--> TYPE I
200 Type set to I.
--> PORT 219,243,169,50,9,226
200 PORT command successful.
--> RETR fichier.txt
150 Binary data connection for fichier.txt (219.243.169.50,2530) 
(10876 bytes).
226 Binary Transfer complete.
10876 bytes received in 0.013 seconds (7.9e+02 Kbytes/s)
ftp> quit

Note the PORT command, PORT 219,243,169,50,9,226. This says the client is listening on IP address 219.243.169.50, port 2530 = (9<<8)+226; the final two integers in the comma-separated list are the 16-bit port number represented as two 8-bit bytes, most significant byte first. The server response beginning with "150" confirms establishment of the data connection to that socket. What isn't shown is that the source port of that connection is always the standard FTP data port, port 20 (remember that FTP servers listen for incoming control connections on port 21).

There are two important points to note about this process:

- The data connection socket is chosen on the fly by the client. This prevents forwarding, since you can't know the port number ahead of time to forward it with SSH. You can get around this problem by establishing the FTP process "by hand" using telnet. That is, choose a data socket beforehand and forward it with SSH, telnet to the FTP server yourself, and issue all the necessary FTP protocol commands by hand, using your forwarded port in the PORT command. But this can hardly be called convenient.
- Remember that the data connection is made in the reverse direction from the control connection; it goes from the server back to the client. As we discussed earlier in this chapter, the usual workaround is to use passive mode.

11.2.5.2 Passive mode in depth

Recall that in a passive-mode transfer, the client initiates a connection to the server. Specifically, instead of listening on a local socket and issuing a PORT command to the server, the client issues a PASV command. In response, the server selects a socket on its side to listen on and reveals it to the client in the response to the PASV command. The client then connects to that socket to form the data connection, and issues the file-transfer command over the control connection. With command line-based clients, the usual way to do passive-mode transfers is to use the passive command. Again, an example:
$ ftp -d aaor.lionaka.net
Connected to aaor.lionaka.net.
220 aaor.lionaka.net FTP server (SunOS 5.7) ready.
----> USER res
331 Password required for res.
Password:
----> PASS XXXX
230 User res logged in.
----> SYST
215 UNIX Type: L8 Version: SUNOS
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> passive
Passive mode on.
ftp> ls
----> PASV
227 Entering Passive Mode (219,243,169,52,128,73)
----> LIST
150 ASCII data connection for /bin/ls (219.243.169.50,2538) (0 bytes).
total 360075
drwxr-xr-x 98 res 500 7168 May  5 17:13 .
dr-xr-xr-x  2 root root  2 May  5 01:47 ..
-rw-rw-r--  1 res 500 596 Apr 25 1999 .FVWM2-errors
-rw--------  1 res 500 332 Mar 24 01:36 .ICEauthority
-rw---------  1 res 500 50 May  5 01:45 .Xauthority
-rw-r--r--  1 res 500 1511 Apr 11 00:08 .Xdefaults
226 ASCII Transfer complete.
ftp> quit
----> QUIT
221 Goodbye.

Note that after the user gives the ls command, the client sends PASV instead of PORT. The server responds with the socket on which it will listen. The client issues the LIST command to list the contents of the current remote directory, and connects to the remote data socket; the server accepts and confirms the connection, then transfers the directory listing over the new connection.

An interesting historical note, which we alluded to earlier, is that the PASV command wasn't originally intended for this use; it was designed to let an FTP client direct a file transfer between two remote servers. The client makes control connections to two remote servers, issues a PASV command to one causing it to listen on a socket, issues a PORT command to the other telling it to connect to the other server on that socket, then issues the data-transfer command (STOR, RETR, etc.). These days, most people don't even know this is possible, and will pull a file from one server to the local machine, and transfer it again to get it to the second remote machine. It's so uncommon that many FTP clients don't support this mode, and some servers prevent its use for security reasons. [Section 11.2.2.3]

11.2.5.3 FTP with the default data ports

The third file-transfer mode occurs if the client issues neither a PORT nor a PASV command. In this case, the server initiates the data connection from the well-known ftp-data port (20) to the source socket of the control connection, on which the client must be listening (these sockets are the "default data ports" for the FTP session). The usual way to use this mode is with the FTP client command sendport, which switches on and off the client's feature of using a PORT command for each data transfer. For this mode, we want it turned off, and it is generally on by default. So the sequence of steps is this:

1. The client initiates the control connection from local socket C to server:21.

322
2. The user gives the sendport command, and then a data-transfer command, such as put or ls. The FTP client begins listening on socket C for an incoming TCP connection.

3. The server determines the socket C at the other end of the control connection. It doesn't need the client to send this explicitly via the FTP protocol, since it can just ask TCP for it (e.g., with the getpeername() sockets API routine). It then opens a connection from its ftp-data port to C, and sends or receives the requested data over that connection.

Now, this is certainly a simpler way of doing things than using a different socket for each data transfer, and so it begs the question of why PORT commands are the norm. If you try this out, you will discover why. First off, it might fail on the client side with the message "bind: Address already in use". And even if it does work, it does so only once. A second ls elicits another address-related error, this time from the server:

```
aaor% ftp syrinx.lionaka.net
Connected to syrinx.lionaka.net.
220 syrinx.lionaka.net FTP server (Version wu-2.5.0(1) Tue Sep 21 16:48:12 EDT
331 Password required for res.
Password:
230 User res logged in.
ftp> sendport
Use of PORT cmds off.
ftp> ls
150 Opening ASCII mode data connection for file list.
keep
fichier.txt
226 Transfer complete.
19 bytes received in 0.017 seconds (1.07 Kbytes/s)
ftp> ls
425 Can't build data connection: Cannot assign requested address.
ftp> quit
```

These problems are due to a technicality of the TCP protocol. In this scenario, every data connection is between the same two sockets, server:ftp-data and C. Since a TCP connection is fully specified by the pair of source and destination sockets, these connections are indistinguishable as far as TCP is concerned; they are different incarnations of the same connection and can't exist at the same time. In fact, to guarantee that packets belonging to two different incarnations of a connection aren't confused, there's a waiting period after one incarnation is closed, during which a new incarnation is forbidden. In the jargon of TCP, on the side that performed an "active close" of the connection, the connection remains in a state called TIME_WAIT. This state lasts for a period that is supposed to be twice the maximum possible lifetime of a packet in the network (or "2MSL", for two times the Maximum Segment Lifetime). After that, the connection becomes fully closed, and another incarnation can occur. The actual value of this timeout varies from system to system, but is generally in the range of 30 seconds to 4 minutes.\[6\]

\[6\] See TCP/IP Illustrated, Volume 1: The Protocols, by W. Richard Stevens (Addison-Wesley), for more technical information about the TIME_WAIT state.

As it happens, some TCP implementations enforce even stronger restrictions. Often, a port that is part of a socket in the TIME_WAIT state is unavailable for use, even as part of a connection to a different remote socket. We have also run into systems that disallow listening on a socket that is currently an endpoint of some connection, regardless of the connection state. These restrictions aren't required by the TCP protocol, but they are common. Such systems usually provide a way to avoid the restrictions, such as the SO_REUSEADDR option of the Berkeley sockets API. An FTP client generally uses this feature, of course, but it doesn't always work!
This address-reuse problem comes up in two places in a default-port FTP transfer. The first one is when the client must start listening on its default data port, which by definition is currently the local endpoint of its control connection. Some systems simply don't allow this, even if the program requests address reuse; that's why the attempt might fail immediately with the message, "address already in use."

The other place is on a second data transfer. When the first transfer is finished, the server closes the data connection, and that connection on the server side moves into the TIME_WAIT state. If you try another data transfer before the 2MSL period has elapsed, the server tries to set up another incarnation of the same connection, and it will fail saying "cannot assign requested address." This happens regardless of the address reuse setting, since the rules of the TCP require it. You can transfer a file again within a few minutes, of course, but most computer users aren't good at waiting a few seconds, let alone minutes. It is this problem that prompts the use of a PORT command for every transfer; since one end of the connection is different every time, the TIME_WAIT collisions don't occur.

Because of these problems, the default-port transfer mode isn't generally used. It has, however, an important property for us: it is the only mode in which the data connection destination port is fixed and knowable before the data-transfer command is given. With this knowledge, some patience, and fair amount of luck, it is possible to forward your FTP data connections through SSH.

### 11.2.6 Forwarding the Data Connection

With all the foregoing discussion in mind, here we simply state the sequence of steps to set up data-connection forwarding. The tricky part is that SSH must request address reuse from TCP for forwarded ports. SSH2 and OpenSSH do this already, but SSH1 can't. It is an easy source modification to make for SSH1, though. In the routine _channel_request_local_forwarding_ in _newchannels.c_, add the following code right before the call to _bind_( ) (in Version 1.2.27, this is at line 1438):

```c
...  
sin.sin_port = htons(port);
{
  int flag = 1;
  setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, (void *)&flag,
  sizeof(flag));
}
/* Bind the socket to the address. */
if (bind(sock, (struct sockaddr *)&sin, sizeof(sin)) < 0)
  packet_disconnect("bind: %.100s", strerror(errno));
...  
```

Recompile and reinstall _sshd_ on the server side. If you're not in a position to do that, you can copy your modified _sshd_ client program to the server, and in the upcoming step (3), use _ssh -L_ from the server to the client instead of _ssh -R_ from client to server.

Another restriction is that the operating system in which the FTP client is running must allow a process to listen on a socket already in use as the endpoint of an existing connection. Some don't. To test this, try an FTP data transfer on the default data ports without SSH, just by using _ftp_ as usual but giving the _sendport_ command before _ls, get_, or whatever. If you get:

```
ftp: bind: Address already in use
```
then your operating system probably won't cooperate. There may be a way to alter this behavior; check the operating system documentation. Figure 11-7 illustrates the following steps.

**Figure 11.7. Forwarding the FTP data connection**

1. Start an SSH connection to forward the control channel as shown earlier in this chapter, and connect with the FTP client. Make sure that passive mode is off.

   ```
   client% ssh1 -f -n -L2001:localhost:21 server sleep 10000 &
   ```
   
or for SSH2:

   ```
   client% ssh2 -f -n -L2001:localhost:21 server
   ```
   
   Then:

   ```
   client% ftp localhost 2001
   Connected to localhost
   220 server FTP server (SunOS 5.7) ready.
   Password: res
   230 User res logged in.
   ftp> sendport
   Use of PORT cmds off.
   ftp> passive
   Passive mode on.
   ftp> passive
   Passive mode off.
   ```
   
   Note that we are using localhost as the forwarding target here, despite our earlier advice. That's OK, because there won't be any PORT or PASV commands with addresses that can be wrong.

2. Now, we need to determine the real and proxy default data ports for the FTP client. On the client side, you can do this with `netstat`:

   ```
   client% netstat -n | grep 2001
   ```
This shows that the source of the control connection from the FTP client to SSH is port 3175. You can do the same thing on the server side, but this time you need to know what's connected to the FTP server port (netstat -n | egrep \`<21\>`), and there may be many things connected to it. If you have a tool like lsrf, it's better to find out the pid of the ftpd or sshd serving your connection and use lsrf -p <pid> to find the port number. If not, you can do a netstat before connecting via FTP and then one right afterward, and try to see which is the new connection. Let's suppose you're the only one using the FTP server, and you get it this way:

```
server% netstat | grep ftp
tcp 0 0 server:32714 server:ftp ESTABLISHED
```

So now, we have the FTP client's default data port (3175), and the source port of the forwarded control connection to the FTP server (32714), which we'll call the proxy default data port; it is what the FTP server thinks is the client's default data port.

3. Now, forward the proxy default data port to the real one:

```
# SSH1, OpenSSH
client% ssh1 -f -n -R32714:localhost:3175 server sleep 10000 &

# SSH2 only
client% ssh2 -f -R32714:localhost:3175 server
```

If, as we mentioned earlier, you don't replace sshd or run a second one, then you'd use the modified ssh on the server in the other direction, like this:

```
server% ./ssh -f -n -L32714:localhost:3175 client sleep 10000 &
```

4. Now, try a data-transfer command with ftp. If all goes well, it should work once, then fail with this message from the FTP server:

```
425 Can't build data connection: Address already in use.
```

(Some FTP servers return that error immediately; others will retry several times before giving up, so it may take a while for that error to appear.) If you wait for the server's 2MSL timeout period, you can do another single data transfer. You can use netstat to see the problem and track its progress:

```
server% netstat | grep 32714
127.0.0.1.32714 127.0.0.1.21 32768 0 32768 0 ESTABLISHED
127.0.0.1.21 127.0.0.1.32714 32768 0 32768 0 ESTABLISHED
127.0.0.1.20 127.0.0.1.32714 32768 0 32768 0 ESTABLISHED
```

The first two lines show the established control connection on port 21; the third one shows the old data connection to port 20, now in the TIME_WAIT state. When that disappears, you can do another data transfer command.

And there you have it: you have forwarded an FTP data connection through SSH. You have achieved the Holy Grail of FTP with SSH, though perhaps you agree with us and Sir Gawain that "it's only a model." Still, if you're terribly concerned about your data connections, have no other way to transfer files, can afford to wait a few minutes between file transfers, and are quite lucky, then this will work. It also makes a great parlor trick at geek parties.
11.3 Pine, IMAP, and SSH

Pine is a popular, Unix-based email program from the University of Washington (http://www.washington.edu/pine/). In addition to handling mail stored and delivered in local files, Pine also supports IMAP\[7] for accessing remote mailboxes and SMTP\[8] for posting mail.

\[7\] Internet Message Access Protocol, RFC-2060.

\[8\] Simple Mail Transfer Protocol, RFC-821.

In this case study, we integrate Pine and SSH to solve two common problems:

**IMAP authentication**

In many cases, IMAP permits a password to be sent in the clear over the network. We discuss how to protect your password using SSH, but (surprisingly) not by port forwarding.

**Restricted mail relaying**

Many ISPs permit their mail and news servers to be accessed only by their customers. In some circumstances, this restriction may prevent you from legitimately relaying mail through your ISP. Once again, SSH comes to the rescue.

We also discuss wrapping ssh in a script to avoid Pine connection delays and facilitate access to multiple mailboxes. This discussion will delve into more detail than the previous one on Pine/SSH integration. [Section 4.5.4]

11.3.1 Securing IMAP Authentication

Like SSH, IMAP is a client/server protocol. Your email program (e.g., Pine) is the client, and an IMAP server process (e.g., imapd) runs on a remote machine, the IMAP host, to control access to your remote mailbox. Also like SSH, IMAP generally requires you to authenticate before accessing your mailbox, typically by password. Unfortunately, in many cases this password is sent to the IMAP host in the clear over the network; this represents a security risk (see Figure 11-8).\[9]

\[9\] IMAP does support more secure methods of authentication, but they aren’t widely deployed.

**Figure 11.8. A normal IMAP connection**

If you have an account on the IMAP host, and if it is running an SSH server, you can protect your password. Because IMAP is a TCP/IP-based protocol, one approach is to use SSH port forwarding between the machine running Pine and the IMAP host (see Figure 11-9). [Section 9.2.1]
However, this technique has two drawbacks:

**Security risk**

On a multiuser machine, any other user can connect to your forwarded port. [Section 9.2.4.3](#) If you use forwarding only to protect your password, this isn't a big deal, since at worst, an interloper could access a separate connection to the IMAP server having nothing to do with your connection. On the other hand, if port forwarding is permitting you to access an IMAP server behind a firewall, an interloper can breach the firewall by hijacking your forwarded port, a more serious security risk.

**Inconvenience**

In this setup, you must authenticate twice: first to the SSH server on the IMAP host (to connect and to create the tunnel) and then to the IMAP server by password (to access your mailbox). This is redundant and annoying.

Fortunately, we can address both these drawbacks and run Pine over SSH securely and conveniently.

### 11.3.1.1 Pine and preauthenticated IMAP

The IMAP protocol defines two modes in which an IMAP server can start: normal and preauthenticated (see Figure 11-10). Normally, the server runs with special privileges to access any user's mailbox, and hence it requires authentication from the client. Unix-based IMAP servers enter this mode when started as root.

**Figure 11.10. Pine/IMAP over SSH, preauthenticated**
Here's a sample session that invokes an IMAP server, *imapd*, through *inetd* so it runs as root:

```plaintext
server% telnet localhost imap
* OK localhost IMAP4rev1 v12.261 server ready
0 login res password'
1 select inbox
* 3 EXISTS
* 0 RECENT
* OK [UIDVALIDITY 964209649] UID validity status
* OK [UIDNEXT 4] Predicted next UID
* FLAGS (\Answered \Flagged \Deleted \Draft \Seen)
* OK [PERMANENTFLAGS (\* \Answered \Flagged \Deleted \Draft \Seen)] Permanent flags
1 OK [READ-WRITE] SELECT completed
2 logout
* BYE imap.example.com IMAP4rev1 server terminating connection
2 OK LOGOUT completed
```

Alternatively, in preauthenticated mode, the IMAP server assumes that authentication has already been done by the program that started the server and that it already has the necessary rights to access the user's mailbox. If you invoke *imapd* on the command line under a nonroot uid, *imapd* assumes you have already authenticated and opens your email inbox. You can then type IMAP commands and access your mailbox without authentication:

```plaintext
server% /usr/local/sbin/imapd
* PREAUTH imap.example.com IMAP4rev1 v12.261 server ready
0 select inbox
* 3 EXISTS
* 0 RECENT
* OK [UIDVALIDITY 964209649] UID validity status
* OK [UIDNEXT 4] Predicted next UID
* FLAGS (\Answered \Flagged \Deleted \Draft \Seen)
* OK [PERMANENTFLAGS (\* \Answered \Flagged \Deleted \Draft \Seen)] Permanent flags
0 OK [READ-WRITE] SELECT completed
1 logout
* BYE imap.example.com IMAP4rev1 server terminating connection
2 OK LOGOUT completed
```

Notice the *PREAUTH* response at the beginning of the session, indicating pre-authenticated mode. It is followed by the command *select inbox*, which causes the IMAP server implicitly to open the inbox of the current user without demanding authentication.

Now, how does all this relate to Pine? When instructed to access an IMAP mailbox, Pine first attempts to log into the IMAP host using *rsh* and to run a preauthenticated instance of *imapd* directly. If this succeeds, Pine then converses with the IMAP server over the pipe to *rsh* and has automatic access to the user's remote inbox without further authentication. This is a good idea and very convenient; the only problem is that *rsh* is very insecure. However, you can make Pine use *SSH* instead.

### 11.3.1.2 Making Pine use SSH instead of *rsh*
Pine’s rsh feature is controlled by three configuration variables in the ~/.pinerc file: rsh-path, rsh-command, and rsh-open-timeout. rsh-path stores the program name for opening a Unix remote shell connection. Normally it is the fully qualified path to the rsh executable (e.g., /usr/ucb/rsh). To make Pine use SSH, instruct it to use the ssh client rather than rsh, setting rsh-path to the location of the SSH client:

rsh-path=/usr/local/bin/ssh

rsh-command represents the Unix command line for opening the remote shell connection: in this case, the IMAP connection to the IMAP host. The value is a printf-style format string with four “%s” conversion specifications that are automatically filled in at runtime. From first to last, these four specifications stand for:

1. The value of rsh-path
2. The remote hostname
3. The username for accessing your remote mailbox
4. The connection method; in this case, "imap"

For example, the default value of rsh-command:

"%s %s -l %s exec /etc/r%sd"

which can instantiate to:

/usr/ucb/rsh imap.example.com -l smith exec /etc/rimapd

To make this work properly with ssh, modify the default format string slightly, adding the -q option for quiet mode:

rsh-command="%s %s -q -l %s exec /etc/r%sd"

This instantiates to:

/usr/local/bin/ssh imap.example.com -w -l smith exec /etc/rimapd

The -q option is necessary so that ssh doesn't emit diagnostic messages that may confuse Pine, such as:

Warning: Kerberos authentication disabled in SUID client.
fwd connect from localhost to local port sshdfwd-2001

Pine otherwise tries to interpret these as part of the IMAP protocol. The default IMAP server location of /etc/r%sd becomes /etc/rimapd.

The third variable, rsh-open-timeout, sets the number of seconds for Pine to open the remote shell connection. Leave this setting at its default value, 15, but any integer greater than or equal to 5 is permissible.

So finally, the Pine configuration is:

rsh-path=/usr/local/bin/ssh
rsh-command="%s %s -q -l %s exec /etc/r%sd"
rsh-open-timeout=
Remote Usernames in Pine

By the way, it's not mentioned in the Pine manpage or configuration file comments, but if you need to specify a different username for connecting to a remote mailbox, the syntax is:

```
{hostname/user=jane}mailbox
```

This causes Pine to call the `rsh-command` with "jane" as the remote username (i.e., the third `%s` substitution).

Generally, you want to use an SSH authentication method that doesn't require typing a password or passphrase, such as trusted-host or public-key with an agent. SSH is run behind the scenes by Pine and doesn't have access to the terminal to prompt you. If you're running the X Window System, `ssh` can pop up an X widget instead to get input, `ssh-askpass`, but you probably don't want that either. Pine may make several separate IMAP connections in the course of reading your mail, even if it's all on the same server. This is just how the IMAP protocol works.

With the previous settings in your `~/.pinerc` file and the right kind of SSH authentication in place, you're ready to try Pine over SSH. Just start Pine and open your remote mailbox; if all goes well, it will open without prompting for a password.

11.3.2 Mail Relaying and News Access

Pine uses IMAP to read mail but not to send it. For that, it can either call a local program (such as `sendmail`) or use an SMTP server. Pine can also be a newsreader and use NNTP (the Network News Transfer Protocol, RFC-977) to contact a news server.

An ISP commonly provides NNTP and SMTP servers for its customers when connected to the ISP's network. However, for security and usage control reasons, the ISP generally restricts this access to connections originating within its own network (including its own dial-up connections). In other words, if you're connected to the Internet from elsewhere and try to use your ISP's services, the attempt will probably fail. Access to your usual servers can be blocked by a firewall, or if not, your outgoing mail can bounce with a message about "no relaying," and the news server rejects you with a message about "unauthorized use."

You are authorized to use the services, of course, so what do you do? Use SSH port forwarding! By forwarding your SMTP and NNTP connections over an SSH session to a machine inside the ISP's network, your connections appear to come from that machine, thus bypassing the address-based restrictions. You can use separate SSH commands to forward each port:

```
$ ssh -L2025:localhost:25 smtp-server ...
$ ssh -L2119:localhost:119 nntp-server ...
```

Alternatively, if you have a shell account on one of the ISP's machines running SSH but can't log into the mail or news servers directly, do this:

```
$ ssh -L2025:smtp-server:25 -L2119:nntp-server:119 shell-server ...
```

This is an off-host forwarding, and thus the last leg of the forwarded path isn't protected by SSH. Section 9.2.4 But since the reason for this forwarding isn't so much protection as it is bypassing the source-address restriction, that's OK. Your mail messages and news postings are going to be transferred insecurely once you drop them off, anyway. (If you want security for them, you need to sign or encrypt them separately, e.g., with PGP or S/MIME.)
In any case, now configure Pine to use the forwarded ports by setting the `smtp-server` and `nntp-server` configuration options in your `~/.pinerc` file:

```plaintext
smtp-server=localhost:2025
nntp-server=localhost:2119
```

### 11.3.3 Using a Connection Script

The Pine configuration option `rsh-path` can point not only to `rsh` or `ssh`, but also to any other program; most usefully, a script you've written providing any needed customizations. There are a couple of reasons why you might need to do this:

- The `rsh-path` setting is global, applying to every remote mailbox. That is, Pine tries to use this style of access either for every remote mailbox or for none. If you have multiple remote mailboxes but only some of them are accessible via SSH/`imapd`, this leads to annoyance. Pine falls back to a direct TCP connection if SSH fails to get an IMAP connection, but you have to wait for it to fail. If the server in question is behind a firewall silently blocking the SSH port, this can be a lengthy delay.
- The “multiple forwarding” problem. You might think to add forwarding options to Pine's `rsh-path` command, rather than run a separate SSH session to get them:

  ```plaintext
  rsh-command="%s %s -q -l %s -L2025:localhost:25 exec /etc/r%sd"
  ```

  This solution can get tricky if you're accessing multiple mailboxes, not only because the command is run for every mailbox, but also because it may run multiple times concurrently. Once the forwarded ports are already established, subsequent invocations will fail. More specifically, SSH1 and OpenSSH will fail altogether; SSH2 issues a warning but continues.

A custom connection script can solve these and other problems. The following Perl script examines the target server and returns failure immediately if it isn't among a small set of known names. This means that Pine moves quickly past the `rsh-path` command for other servers and attempts a direct IMAP connection. The script also discovers whether SMTP and NNTP forwarding are in place, and includes those in the SSH command only if they aren't. To use this script or another like it, point Pine's `rsh-path` option to your script, and set `rsh-command` to be compatible with your script:

```plaintext
rsh-path=/path/to/script
rsh-command=%s %s %s %s
```

Here is a sample implementation of the script, using Perl:

```perl
#!/usr/bin/perl

# TCP/IP module
use IO::Socket;

# get the arguments passed by Pine
($server,$remoteuser,$method) = @ARGV;

die "usage: $0 <server> <remote user> <method>"
    unless scalar @ARGV == 3;

if ($server eq "mail.isp.com") {
    # on this machine, I had to compile my own imapd
    $command = 'cd ~/bin; exec imapd';
} elseif ($server eq "clueful.isp.com") {
    # on this box, the POP and IMAP servers are in the expected place
```
$command = "exec /etc/r${method}d";
} else {
    # signal Pine to move on
    exit 1;
}

smtp = 25;  # well-known port for SMTP
nntp = 119;  # and NNTP
smtp_proxy = 2025;  # local port for forwarding SMTP connection
nntp_proxy = 2119;  # local port for forwarding NNTP connection
ssh = '/usr/local/bin/ssh1';  # which SSH do I want to run?

# Try to connect to the forwarded SMTP port; only do forwarding if the
# attempt fails. Also, do forwarding only if we're not in the domain
# "home.net". The idea is that that's your home network, where you have
# direct access to your ISP's mail and news servers.
do_forwards = !defined($socket = IO::Socket::INET->
    >new("localhost:$smtp_proxy"))
    && `domainname` !~ /HOME.NET/i;

# be tidy
close $socket if $socket;

# Set the forwarding options if we're doing forwarding. This assumes that
# the mail and news servers are called "mail" and "news", respectively, in
# your ISP's domain; a common and useful convention.
forward = ('-L',"smtp_proxy:mail:$smtp","-L","nntp_proxy:news:$nntp");
if ($do_forwards);

# prepare the arguments to ssh
ssh_argv = ('-a','-x','-q',@forward,"$remoteuser@$server");

# run ssh
exec ssh, @ssh_argv, $command;

11.4 Kerberos and SSH

Kerberos is an authentication system designed to operate securely in an environment where
networks may be monitored and user workstations aren't under central control. [Section 1.6.3] It
was developed as part of Project Athena, a wide-ranging research and development effort carried
out at MIT between 1983 and 1991, funded primarily by IBM and Digital Equipment Corporation.
Project Athena contributed many other pieces of technology to the computing world, including
the well-known X Window System.

Kerberos is very different in character and design from SSH; each includes features and services
the other lacks. In this study, we compare the two systems in detail, and then discuss how to
combine them to obtain the advantages of both. If your site already uses Kerberos, you can add
SSH while maintaining your existing account base and authentication infrastructure. (Figure 11-11
shows where Kerberos fits into the scheme of SSH configuration.) If you aren't using Kerberos, its
advantages may also be compelling enough to motivate you to install it, especially for large
computing environments.

Figure 11.11. Kerberos configuration (highlighted parts)
There are two versions of the Kerberos protocol, Kerberos-4 and Kerberos-5. Free reference implementations of both are available from MIT:

ftp://athena-dist.mit.edu/pub/kerberos/

Kerberos-5 is the current version, and Kerberos-4 is no longer actively developed at MIT. Even so, Kerberos-4 is still in use in many contexts, especially bundled into commercial systems (e.g., Sun Solaris, Transarc AFS). SSH1 supports Kerberos-5, and OpenSSH/1 supports Kerberos-4. The current draft of the SSH-2 protocol doesn't yet define a Kerberos authentication method, but as this book went to press SSH 2.3.0 was released with "experimental" Kerberos-5 support, which we haven't covered here (but which should work substantially as described with SSH1).

11.4.1 Comparing SSH and Kerberos

While they solve many of the same problems, Kerberos and SSH are very different systems. SSH is a lightweight, easily deployed package, designed to work on existing systems with minimal changes. Kerberos, in contrast, requires you to establish a significant infrastructure before use.

11.4.1.1 Infrastructure

Let's consider an example: allowing users to create secure sessions between two machines. With SSH, you simply install the SSH client on the first machine and the server on the second, start the server, and you're ready to go. Kerberos, however, requires the following administrative tasks:

- Establish at least one Kerberos Key Distribution Center (KDC) host. The KDCs are central to the Kerberos system and must be heavily secured; typically they run nothing but the KDC, don't allow remote login access, and are kept in a physically secure location. [110] Kerberos can't operate without a KDC, so it is wise to establish backup or "slave" KDCs also, which then must be synchronized periodically with the master. A KDC host might also run a remote administration server, a credentials-conversion server for Kerberos-4 compatibility in a Kerberos-5 installation, and other server programs depending on your needs.
For each Kerberos user, add an account (or "user principal") to the KDC database.

For each application server that will use Kerberos to authenticate clients, add an account (or "service principal") to the KDC database. A separate principal is required for each server on each host.

Distribute the service principal cryptographic keys in files on their respective hosts.

Write a sitewide Kerberos configuration file (/etc/krb5.conf) and install it on all hosts.

Install Kerberos-aware applications. Unlike SSH, Kerberos isn't transparent to TCP applications. For example, you might install a version of telnet that uses Kerberos to provide a strongly authenticated, encrypted remote login session similar to that of ssh.

Deploy a clock-synchronization system such as Network Time Protocol (NTP). Kerberos relies on timestamps for proper operation.

Clearly, deploying Kerberos requires much more work and more changes to existing systems than SSH does.

11.4.1.2 Integrating with other applications

Another difference between SSH and Kerberos is their intended use. SSH is a set of programs that work together via the SSH protocol, designed to use in combination with existing applications with minimal changes. Consider programs like CVS [Section 8.2.6.1] and Pine [Section 11.3] that invoke the insecure rsh program internally to run remote programs. If configured to use ssh instead of rsh, the program's remote connections become secure; the introduction of ssh is transparent to the program and its remote partner. Alternatively, if an application makes a direct network connection to a TCP service, SSH port forwarding can secure that connection simply by telling the application to use a different server address and port.

Kerberos, on the other hand, is designed as an authentication infrastructure, together with a set of programming libraries. The libraries are for adding Kerberos authentication and encryption to existing applications; this process is called kerberizing the application. The MIT Kerberos distribution comes with a set of common, kerberized services, including secure versions of telnet, ftp, rsh, su, etc.

SSH2 has recently moved towards this model. It is similarly organized as a set of libraries implementing the SSH-2 protocol, accessed via an API by client and server programs.

11.4.1.3 Security of authenticators

The extra complexity of Kerberos provides properties and capabilities that SSH doesn't. One major win of Kerberos is its transmission and storage of authenticators (i.e., passwords, secret keys, etc.). To demonstrate this advantage, let's compare Kerberos's ticket system with SSH's password and public-key authentication.

SSH password authentication requires your password each time you log in, and it is sent across the network each time. The password isn't vulnerable during transmission, of course, since SSH encrypts the network connection. However, it does arrive at the other side and exist in plaintext inside the SSH server long enough for authentication to occur, and if the remote host has been compromised, an adversary has an opportunity to obtain your password.

SSH cryptographic authentication, on the other hand, may require you to store your private keys on each client host, and you must have authorization files in each server account you want to access. This presents security and distribution problems. A stored key is protected by encryption with a passphrase, but having it stored at all on generally accessible hosts is a weakness Kerberos doesn't have. An adversary who steals your encrypted key may subject it to an offline dictionary attack to try to guess your passphrase. If successful, your adversary has access to your accounts...
until you notice and change all your key and authorization files. This change may be time-consuming and error-prone if you have several accounts on different machines, and if you miss one, you're in trouble.

Kerberos ensures that a user's password\footnote{Actually, the secret key derived from the user's password, but the distinction isn't relevant here.} travels as little as possible and is never stored outside the KDC. When a user identifies herself to the Kerberos system, the identifying program (\textit{kinit}) uses her password for an exchange with the KDC, then immediately erases it, never having sent it over the network in any form nor stored it on disk. A client program that subsequently wants to use Kerberos for authentication sends a "ticket," a few bytes of data cached on disk by \textit{kinit}, which convinces a kerberized server of the user's identity. Tickets are cached in files readable only by their users, of course, but even if they are stolen, they are of limited use: tickets expire after a set amount of time, typically a few hours, and they are specific to a particular client/server/service combination.

A stolen Kerberos ticket cache can be the target of a dictionary attack, but with an important difference: user passwords aren't present. The keys in the cache belong to server principals, and moreover, they are typically generated randomly and hence less vulnerable to a dictionary attack than user passwords. Sensitive keys are stored only on the KDCs, under the theory that it is much easier to effectively secure a small set of limited-use machines, rather than a large set of heterogeneous, multipurpose servers and workstations over which the administrator may have little control. Much of Kerberos's complexity results from this philosophy.

\subsection*{11.4.1.4 Account administration}

Kerberos also serves other functions beyond the scope of SSH. Its centralized user account database can unify those of disparate operating systems, so you may administer one set of accounts instead of keeping multiple sets synchronized. Kerberos supports access control lists and user policies for closely defining which principals are allowed to do what; this is authorization, as opposed to authentication. Finally, a Kerberos service area is divided into realms, each with its own KDC and set of user accounts. These realms can be arranged hierarchically, and administrators can establish trust relationships between parent/child or peer realms, allowing automatic cross-authentication between them.

\subsection*{11.4.1.5 Performance}

Kerberos authentication is generally faster than SSH public-key authentication. This is because Kerberos usually employs DES or 3DES, whereas SSH uses public-key cryptography, which is much slower in software than any symmetric cipher. This difference may be significant if your application needs to make many short-lived secure network connections and isn't running on the fastest hardware.

To sum up: Kerberos is a system of broader scope than SSH, providing authentication, encryption, key distribution, account management, and authorization services. It requires substantial expertise and infrastructure to deploy and requires significant changes to an existing environment for use. SSH addresses fewer needs, but has features that Kerberos installations typically don't, such as port forwarding. SSH is much more easily and quickly deployed and is more useful for securing existing applications with minimal impact.

\subsection*{11.4.2 Using Kerberos with SSH}

Kerberos is an authentication and authorization (AA) system. SSH is a remote-login tool that performs AA as part of its operation, and one AA system it can use is (you guessed it) Kerberos. If
your site already uses Kerberos, its combination with SSH is compelling, since you can apply your existing infrastructure of principals and access controls to SSH.

Even if you're not already using Kerberos, you might want to roll it out together with SSH as an integrated solution because of the advantages Kerberos provides. By itself, the most flexible SSH authentication method is public-key with an agent. Passwords are annoying and limited because of the need to type them repeatedly, and the trusted-host method isn't appropriate or secure enough for many situations. Unfortunately, the public-key method incurs substantial administrative overhead: users must generate, distribute, and maintain their keys, as well as manage their various SSH authorization files. For a large site with many nontechnical users, this can be a big problem, perhaps a prohibitive one. Kerberos provides the key-management features SSH is missing. SSH with Kerberos behaves much like public-key authentication: it provides cryptographic authentication that doesn't give away the user's password, and the ticket cache gives the same advantages as the key agent, allowing for single sign-on. But there are no keys to generate, authorization files to set up, or configuration files to edit; Kerberos takes care of all this automatically.

There are some disadvantages. First of all, only the Unix SSH packages have Kerberos support; we know of no Windows or Macintosh products containing it. Only the SSH-1 protocol currently supports Kerberos, although there is work in progress in the SECSH working group to add Kerberos to SSH-2. Second, public-key authentication is tied to other important features of SSH, such as forced commands in the authorization file, that can't be used with Kerberos authentication. This is an unfortunate artifact of the way Unix SSH has evolved. Of course, you can still use public-key authentication as needed. You may find the access controls of Kerberos adequate for most needs and use public-key for a few situations in which you need finer-grained control.

In the following sections, we explain how to use the SSH Kerberos support. If your site has a kerberized SSH installed, this should be enough to get you going. We can't discuss all the gory detail of building a Kerberos infrastructure, but we do give a quick outline of how to set up Kerberos from scratch, if you have your own systems and want to try it. However, these are just hints, and the description is incomplete. If you're going to use, install, and manage kerberized SSH, you need a more complete understanding of Kerberos than you will get here. A good place to start is:

http://web.mit.edu/kerberos/www/

11.4.3 A Brief Introduction to Kerberos-5

In this section, we introduce the important concepts of principals, tickets, and ticket-granting-tickets (TGTs), and follow them with a practical example.

11.4.3.1 Principals and tickets

Kerberos can authenticate a user or a piece of software providing or requesting a service. These entities have names, called principals, that consist of three parts: a name, an instance, and a realm, notated as name/instance@REALM. Specifically:

- The name commonly corresponds to a username for the host operating system.
- The instance, which may be null, typically distinguishes between the same name in different roles. For example, the user res might have a normal, user-level principal res@REALM (note the null instance), but he could have a second principal with special privileges, res/admin@REALM, for his role as a system administrator.

[13] This was the case in Kerberos-4. In fact, Kerberos-5 principals have a realm, plus any number of "components"—the first two of which are conventionally used as the name and instance, as in Kerberos-4.
The realm is an administrative division identifying a single instance of the Kerberos principal database (that is, a list of principals under common administrative control). Each host is assigned a realm, and this identification is relevant to authorization decisions, which we discuss shortly. Realms are always uppercase, by convention.

As we've discussed, Kerberos is based on tickets. If you want to use a network service—say, the telnet server on a host, to log in remotely—you must obtain a ticket for that service from the Kerberos Key Distribution Center, or KDC. The ticket contains an authenticator, proving your identity to the software providing the service. Since both you and the service must be identified to the KDC, both must have principals.

The system administrator establishes principals by adding them to the KDC database. Each principal has a secret key, known only to the principal owner and to the KDC; the operation of the Kerberos protocol is based on this fact. For instance, when you request a ticket for a service, the KDC gives you some bits that have been encrypted with the secret key of the service. Therefore, only the intended service can decrypt and verify the ticket. Moreover, a successful decryption proves that the KDC issued the ticket, since only the service and the KDC know the service's secret key.

For a user principal, the secret key is derived from the user's Kerberos password. Service principal keys are usually stored in the file /etc/krb5.keytab on the host where the service runs, and the service calls a Kerberos library routine to read the file and extract its secret key. Obviously this file must be protected from general read access, since anyone who can read it can impersonate the service.

### 11.4.3.2 Obtaining credentials with kinit

Let's use an example to get a practical look at Kerberos. Suppose you are on a Unix host spot in the realm FIDO, and you want to use kerberized telnet to log into another host, rover. First, you obtain Kerberos credentials by running the command `kinit`:

```
[res@spot res]$ kinit
Password for res@FIDO : ********
```

`kinit` assumes that since your username is res and the host spot is in the realm FIDO, you want to obtain credentials for the principal res@FIDO. If you had wanted a different principal, you could have supplied it as an argument to `kinit`.

### 11.4.3.3 Listing credentials with klist

Having successfully gotten your credentials with `kinit`, you can examine them with the `klist` command, which lists all tickets you have obtained:

```
[res@spot res]$ klist
Ticket cache: /tmp/krb5cc_84629
Default principal: res@FIDO
Valid starting Expires Service principal
07/09/00 23:35:03 07/10/00 09:35:03 krbtgt/FIDO@FIDO
```

So far, you have only one ticket, for the service krbtgt/FIDO@FIDO. This is your Kerberos TGT, and it is your initial credential: proof to be presented later to the KDC that you have successfully authenticated yourself as res@FIDO. Note that the TGT has a validity period: it expires in 10 hours. After that, you must do another `kinit` to reauthenticate yourself.

### 11.4.3.4 Running a kerberized application
Having gotten your credentials, you now `telnet` to the remote host:

```
[res@spot res]$ telnet -a rover
Trying 10.1.2.3...
Connected to rover (10.1.2.3).
Escape character is '^]'.
[Kerberos V5 accepts you as "res@FIDO"]
Last login: Sun Jul  9 16:06:45 from spot
You have new mail.
[res@rover res]$
```

The `-a` option to this kerberized `telnet` client tells it to do auto-login: that is, it attempts to negotiate Kerberos authentication with the remote side. It succeeds: the remote side accepts your Kerberos identification, and allows you to log in without providing a password. If you return to spot and do a `klist`, you will see what happened:

```
[res@spot res]$ klist
Ticket cache: /tmp/krb5cc_84629
Default principal: res@FIDO
Valid starting Expires Service principal
07/09/00 23:35:03 07/10/00 09:35:03 krbtgt/FIDO@FIDO
07/09/00 23:48:10 07/10/00 09:35:03 host/rover@FIDO
```

Note that you now have a second ticket, for the service "host/rover@FIDO". This principal is used for remote login and command execution services on the host rover, such as kerberized `telnet`, `rlogin`, `rsh`, etc. When you ran `telnet -a rover`, the `telnet` client requested a ticket for host/rover@FIDO from the KDC, supplying your TGT with the request. The KDC validated the TGT, verifying that you had recently identified yourself as res@FIDO, and issued the ticket. `telnet` stored the new ticket in your Kerberos ticket cache, so that the next time you connect to rover, you can just use the cached ticket instead of contacting the KDC again (at least, until the ticket expires). It then presented the host/rover@FIDO ticket to the `telnet` server, which verified it and in turn believed that the client had been identified as res@FIDO to the KDC.

### 11.4.3.5 Authorization

So far we've taken care of authentication, but what about authorization? The `telnet` server on rover believes that you are res@FIDO, but why should res@FIDO be allowed to log in? This comes back to the host/realm correspondence we've mentioned. [Section 11.4.3.2] Since you didn't specify otherwise, the `telnet` client told the server that you wanted to log into the account res on rover. (You could have changed that with `telnet -l username`.) Since rover is also in the realm FIDO, Kerberos applies a default authorization rule: if host H is in realm R, the Kerberos principal u@R is allowed access to the account u@H. Using this default rule implies that the system administrators are managing the correspondence between operating system (OS) usernames and Kerberos principals. If you had tried to log into your friend Bob's account instead, here's what would have happened:

```
[res@spot res]$ telnet -a -l bob rover
Trying 10.1.2.3...
Connected to rover (10.1.2.3).
Escape character is '^]'.
[Kerberos V5 accepts you as "res@FIDO"]
telnetd: Authorization failed.
```

Note that authentication was still successful: the `telnet` server accepted you as res@FIDO. The authorization decision failed, though: Kerberos decided that the principal res@FIDO was not allowed to access the account bob@rover. Bob can allow you to log into his account by creating the file `rover:`bob/.k5login, and placing a line in it containing your principal name, res@FIDO.
He would also have to place his own principal in there, since if a .k5login file exists, it overrides the default authorization rule, and Bob would be unable to log into his own account. So Bob's authorization file would look like this:

rover:~bob/.k5login:
   bob@FIDO
   res@FIDO

11.4.4 Kerberos-5 in SSH1

To enable Kerberos support in SSH1, compile it --with-kerberos5. [Section 4.1.5.7] If your Kerberos support files (libraries and C header files) aren't in a standard place and configure can't find them, you can tell it where to look using:

# SSH1 only
$ configure ... --with-kerberos5=/path/to/kerberos ...

Two notes on doing this build:

- The MIT Kerberos-5 Release 1.1 renamed the library libcrypto.a to libk5crypto.a, and the SSH1 build files have not been updated to reflect this. You can either alter the SSH1 Makefile, or just use:

  # cd your_Kerberos_library_directory
  # ln -s libk5crypto.a libcrypto.a

- The routine krb5_xfree(), used in auth-kerberos.c, also appears to have disappeared in 1.1. Replacing all occurrences of krb5_xfree with xfree appears to work.

If you compile in Kerberos support, the resulting SSH programs work only on a system with Kerberos installed, even if you aren’t using Kerberos authentication. The programs will likely refer to Kerberos shared libraries that must be present for the programs to run. Also, SSH performs Kerberos initialization on startup and expects a valid host Kerberos configuration file (/etc/krb5.conf).

After installation, we recommend setting the serverwide configuration keyword KerberosAuthentication in /etc/sshd_config to "yes" for clarity, even though it is on by default:

# SSH1 only
KerberosAuthentication yes

Additionally, the host/server@REALM principal must be in the KDC database, and its key must be stored in /etc/krb5.keytab on the server.

Once running with Kerberos support, SSH1 operates essentially as we described for kerberized telnet; Figure 11-12 illustrates the process. [Section 11.4.3.4] On the client, simply run kinit to obtain your Kerberos TGT, and then try ssh -v. If Kerberos authentication succeeds, you will see:

$ ssh -v server
...
server: Trying Kerberos V5 authentication.
server: Kerberos V5 authentication accepted.
...
and in the server log:

Kerberos authentication accepted joe@REALM for login to account joe from client_host

Figure 11.12. SSH with Kerberos authentication

As with telnet, if you want to allow someone else to log into your account using Kerberos and ssh -l your_username, you must create a ~/.k5login file and place their principal name in it, along with your own.

11.4.4.1 Kerberos password authentication

If Kerberos authentication is enabled in the SSH server, password authentication changes in behavior. Passwords are now validated by Kerberos instead of the host operating system. This behavior is usually desired in a fully kerberized environment, where local passwords might not be usable at all. In a mixed environment, however, it may be useful to have SSH fall back on the operating system (OS) password if Kerberos validation fails. The SSH server option that controls this feature is KerberosOrLocalPasswd:

# SSH1, OpenSSH
KerberosOrLocalPasswd yes

This fallback is useful as a fail-safe: if the KDC isn't functioning, you can still authenticate by your OS password (although public-key would be a stronger failsafe authentication method).

Another feature of kerberized password authentication is that sshd stores your TGT upon login, so you don't need to run kinit and retype your password to get Kerberos credentials on the remote host.

11.4.4.2 Kerberos and NAT

SSH is frequently used across firewalls, and these days such a boundary often includes network address translation. Unfortunately, Kerberos has a serious problem with NAT. Kerberos tickets usually include a list of IP addresses from which they are allowed to be used; that is, the client
presenting the ticket must be transmitting from one of those addresses. By default, \textit{kinit} requests a
TGT limited to the IP addresses of the host it's running on. You can see this with the \textit{-a} option to \textit{klist}:

\begin{verbatim}
[res@spot res]$ klist -a -n
Ticket cache: /tmp/krb5cc_84629
Default principal: res@FIDO

Valid starting     Expires            Service principal
07/09/00 23:35:03  07/10/00 09:35:03  krbtgt/FIDO@FIDO
Addresses: 10.1.2.1
07/09/00 23:48:10  07/10/00 09:35:03  host/rover@FIDO
Addresses: 10.1.2.1
\end{verbatim}

(The \textit{-n} switch tells \textit{klist} to display the addresses by number, rather than translating them to names.)
Host spot's IP address is 10.1.2.1, and so the KDC issues the TGT limited to use from that address.
If spot has multiple network interfaces or addresses, they are listed here as well. When you obtain
subsequent service tickets based on this TGT, they are also limited to the same set of addresses.

Now, imagine you connect to an SSH server on the other side of a NAT gateway, which is
rewriting your (the client's) IP address, but the KDC is inside the NAT boundary, with you.

When you obtain the service ticket from the KDC, it contains your real IP address. The SSH
server, however, sees your NAT'd address as the source of the connection, notes that this doesn't
match the address encoded in the ticket, and refuses authentication. In this case, \textit{ssh -v} reports:

\begin{verbatim}
Trying Kerberos V5 authentication.
Kerberos V5: failure on credentials (Incorrect net address).
\end{verbatim}

\textbf{Figure 11.13} illustrates this problem. It has no good solution at the moment. One workaround is
the undocumented \textit{kinit -A} switch, which causes \textit{kinit} to request a ticket with \textit{no} addresses in it at
all. This trick decreases security, because a stolen ticket cache can then easily be used from
anywhere, but it gets around the problem.

\textbf{Figure 11.13. Kerberos and NAT}

\begin{center}
\includegraphics[width=\textwidth]{fig11_13.png}
\end{center}

\subsection*{11.4.4.3 Cross-realm authentication}
Kerberos realms are distinct collections of principals under separate administrative control. For instance, you might have two departments, Sales and Engineering, that don't trust each other (just for the sake of example, of course). The Sales people don't want any of those weird Engineers to be able to create accounts in their space, and Engineering certainly doesn't want any sales-droids mucking about with their logins. So you create two Kerberos realms, SALES and ENGINEERING, and have their respective administrators deal with account management in each realm.

The catch is, of course, that Sales and Engineering do need to work together. Sales guys need to log into Engineering machines to try out new products, and Engineering needs access to Sales desktops to fix the problems they constantly get into. Suppose Erin the engineer needs Sam from Sales to access her account on an Engineering machine, erin@bulwark. She can place Sam's principal name in her Kerberos authorization file, like so:

```
bulwark:~erin/.k5login:
erin@ENGINEERING
sam@SALES
```

However, this won't work. To log in, Sam needs a service ticket for host/bulwark@ENGINEERING. Only a KDC for the ENGINEERING realm can issue such a ticket, but an ENGINEERING KDC won't know the principal sam@SALES. In general, an ENGINEERING host has no way of authenticating a principal from the SALES realm. It looks as if Sam will need a principal in the ENGINEERING domain as well, but this poor solution violates the whole idea of having separate realms. It's also cumbersome, since Sam would have to do another `kinit` each time he wants to access resources in a different realm.

The solution to this problem is called **cross-realm authentication.** First, both realms must be described in the `/etc/krb5.conf` files on all machines in both realms; Kerberos knows only those realms listed in the configuration file. Then the administrators of the two realms establish a shared secret key between them, called a cross-realm key. The key is realized as a common key for two specially named principals, one in each realm. The key has a direction, and its existence allows one KDC to issue a TGT for the other realm; the other KDC can verify that this TGT was issued by its trusted peer realm using the shared key. With one cross-realm key in place, authentication in one realm provides a verifiable identity in the other realm as well. If the trust is symmetric—that is, if each realm should trust the other—then two cross-realm keys are needed, one for each direction.

**Hierarchical Realms in Kerberos-5**

For a large number of realms, the system as described quickly becomes unwieldy. If you want cross-realm trust between all of them, you must manually establish cross-realm keys for each pair of realms. Kerberos-5 supports hierarchical realms to address this problem. A realm name containing dots, such as ENGINEERING.BIGCORP.COM, implies the (possible) existence of realms BIGCORP.COM and COM. When attempting cross-realm authentication from SALES.BIGCORP.COM to ENGINEERING.BIGCORP.COM, if Kerberos doesn't find a direct cross-realm key, it attempts to navigate up and then down the realm hierarchy, following a chain of cross-realm relations to the target realm. That is, if there are cross-realm keys from SALES.BIGCORP.COM to BIGCORP.COM and from BIGCORP.COM to ENGINEERING.BIGCORP.COM, the cross-realm authentication from SALES to ENGINEERING succeeds without needing an explicit cross-realm setup between them. This allows for scalable, complete, cross-realm relationships among a large collection of realms.

Note that Sam doesn't have a second principal now, sam@ENGINEERING. Rather, an ENGINEERING KDC can now verify that Sam was authenticated as sam@SALES by a SALES
KDC and can therefore use the principal sam@SALES in authorization decisions. When Sam tries to log into bulwark using SSH, Kerberos notices that the target machine is in a different realm from Sam's principal and automatically uses the appropriate cross-realm key to obtain another TGT for him in the ENGINEERING realm. Kerberos then uses that to obtain a service ticket authenticating sam@SALES to host/bulwark@ENGINEERING. sshd on bulwark reads Erin's ~/.k5login file, sees that sam@SALES is allowed access, and permits the login.

That's the basic idea. However, when SSH enters the picture, cross-realm authentication can fail due to a confusing catch. Suppose Sam uses bulwark so often that he's given an account there. The sysadmin puts "sam@SALES" into bulwark:~sam/.k5login, so that Sam can log in there with his SALES credentials. But, it doesn't work. Even with everything set up correctly so that cross-realm kerberized telnet works, SSH Kerberos authentication still fails for him. Even more mysteriously, every other form of authentication starts failing as well. Sam had public-key authentication set up and working before, and you'd expect it to try Kerberos, fail, then try public-key and succeed. But all the public-key attempts fail, too. Sam won't get much of a hint of what the problem is unless password authentication is turned on, and SSH eventually tries it:

```
[sam@salesa sam]$ ssh -v bulwark
...  
Trying Kerberos V5 authentication.
Kerberos V5 authentication failed.
Connection to authentication agent opened.
Trying RSA authentication via agent with 'Sam's personal key'
Server refused our key.
Trying RSA authentication via agent with 'Sam's work key'
Server refused our key.
Doing password authentication.
sam@SALES@bulwarks's password:
```

That last prompt doesn't look right at all: "sam@SALES@bulwark"? There's another hint from sshd -d:

```
Connection attempt for sam@SALES from sales

SSH is mistakenly using the principal name as if it were the account name—as if Sam had typed, ssh -l sam@SALES bulwark. Of course, there's no account named "sam@SALES"; there's one named "sam ". And in fact, the quick fix for this problem is for Sam to specify his username explicitly, with ssh -l sam bulwark, even though this seems redundant.

The reason for this odd problem is a Kerberos-5 feature that SSH employs, called aname → lname (authentication name to local name mapping). Kerberos can be used with a variety of operating systems, some of whose notions of a username don't correspond easily with Kerberos principal names. Perhaps the usernames allow characters that are illegal in principal names, or there are multiple operating systems with conflicting username syntaxes. Or perhaps when you merge two existing networks, you find username conflicts among existing accounts, so that the principal

```
res@REALM
```

must translate to the account res on some systems, but rsilverman on others. The Kerberos-5 designers thought it would be good if Kerberos could automate handling this problem itself, and so they included the aname → lname facility for translating principals to the correct local account names in various contexts.

SSH1 uses aname → lname. When doing Kerberos authentication, the SSH1 client supplies the principal name as the target account name by default, rather than the current local account name (that is, it behaves as if Sam had typed ssh -l sam@SALES bulwark). The server in turn applies the
aname → lname mapping to this, to turn it into a local account name. When the principal name and the server host are in the same realm, this works automatically, because there is a default aname → lname rule that maps

user@REALM
to "user" if REALM is the host's REALM. However, Sam is doing cross-realm authentication, and so the two realms are different: his principal is

sam@SALES
, but the server's realm is ENGINEERING. So the aname → lname mapping fails, and sshd goes ahead with using "sam@SALES" as the local account name. Since there is no account with that name, every form of authentication is guaranteed to fail.

The system administrators of the ENGINEERING realm can fix this problem by configuring an aname → lname mapping for SALES. As it happens, though, the aname → lname facility in MIT Kerberos-5 Release 1.1.1 appears to be unfinished. It's almost entirely undocumented and includes references to utilities and files that don't appear to exist. However, we did manage to uncover enough information to give one example solution. From the comments in the source file src/lib/krb5/os/an_to_ln.c, we devised the following "auth_to_local" statements that can fix Sam's problem:

```
bulwark:/etc/krb5.conf:
...
[realms]
ENGINEERING = {
    kdc = kerberos.engineering.bigcorp.com
    admin_server = kerberos.engineering.bigcorp.com
    default_domain = engineering.bigcorp.com
    auth_to_local = RULE:[1:$1]
    auth_to_local = RULE:[2:$1]
    auth_to_local = DEFAULT
}
```

These rules cause the aname → lname function on this host to map principals of the form foo@REALM or foo/bar@REALM to the username "foo" for all realms, as well as applying the default translation rule for the host's realm.

### 11.4.4.4 TGT forwarding

Recall that Kerberos tickets are normally issued to be usable only from the requesting host. If you do a kinit on spot, then use SSH to log into rover, you are now stuck as far as Kerberos is concerned. If you want to use some Kerberos service on rover, you must run another kinit, because your credentials cache is stored on spot. And it won't help to copy the credentials cache file from spot to rover because the TGT won't be valid there; you need one issued for rover. If you do another kinit, your password is safe traveling over the network through SSH, but this is still not a single sign-on, and it's annoying.

SSH has an analogous problem with public-key authentication and solves it with agent forwarding. [Section 6.3.5] Similarly, Kerberos-5 solves it with TGT forwarding. The SSH client asks the KDC to issue a TGT valid on the server host, based on the client's holding an existing valid TGT. When it receives the new TGT, the client passes it to sshd, which stores it in the remote account's Kerberos credentials cache. If successful, you'll see this message in the output from ssh -v:

```
Trying Kerberos V5 TGT passing.
Kerberos V5 TGT passing was successful.
```
and a klist on the remote host shows the forwarded TGT.

In order to use TGT forwarding, you must compile SSH with the switch `--enable-kerberos-tgt-passing`. You must also request a forwardable TGT with `kinit -f`; otherwise, you see:

```
Kerberos V5 krb5_fwd_tgt_creds failure (KDC can't fulfill requested option)
```

### 11.4.4.5 SSH1 Kerberos ticket-cache bug

Prior to Version 1.2.28, SSH1 had a serious flaw in its Kerberos ticket cache handling. Under some circumstances SSH1 mistakenly set the KRB5CCNAME environment variable on the remote side to the string "none". This variable controls where the ticket cache is stored. The ticket cache contains sensitive information; anyone who steals your ticket cache can impersonate you for the lifetime of its tickets. Normally, the ticket cache file is kept in `/tmp`, which is reliably local to each machine. Setting KRB5CCNAME to `none` means that when the user does a `kinit`, the ticket cache is established in a file named `none` in the current working directory. This directory can easily be an NFS filesystem, allowing the tickets to be stolen by network snooping. Or it can be an inappropriate spot in the filesystem, perhaps one where inherited ACLs give someone else the right to read the file, regardless of the ownership and permissions set by SSH.

```
Don't use Kerberos authentication in SSH1 versions earlier than 1.2.28.
```

This bug was fixed by SSH Communications Security in Version 1.2.28 in response to our bug report. Note that this problem occurs if SSH1 is compiled with Kerberos support, even if Kerberos authentication isn't in use for the session at hand. The OpenSSH Kerberos-4 code has never had this bug.

### 11.4.4.6 Kerberos-5 setup notes

Here we present an abbreviated "quick-start" menu of steps to set up a working, one-host Kerberos system from scratch, using the MIT Kerberos-5 distribution Version 1.1.1. This is far from complete and might be wrong or misleading for some environments or builds. It's just meant to get you started, if you want to give Kerberos a try. Suppose the local host's name is `shag.carpet.net`, and your chosen realm name is `FOO`, and your username is "fred":

1. Compile and install krb5-1.1.1. We complied with `--localstatedir=/var` so the KDC database files go under `/var`.
2. Run:

   ```
   $ mkdir /var/krb5kdc
   ```

3. Install an `/etc/krb5.conf` file as follows. Note the log files; these will be useful to examine later in case of problems (or just for information):

   ```
   [libdefaults]
   ticket_lifetime = 600
   default_realm = FOO
   default_tkt_enctypes = des-cbc-crc
   default_tgs_enctypes = des-cbc-crc
   
   [realms]
   FOO = {
       kdc = shag.carpet.net
       admin_server = shag.carpet.net
   }
   ```

346
default_domain = carpet.net

)[domain_realm]
  .carpet.net = FOO
carpet.net = FOO

)[logging]
  kdc = FILE:/var/log/krb5kdc.log
  admin_server = FILE:/var/log/kadmin.log
default = FILE:/var/log/krb5lib.log

Install a file /var/krb5kdc/kdc.conf like this:

)[kcde_defaults]
  kdc_ports = 88,750

)[realms]
  FOO = {
    database_name = /var/krb5kdc/principal
    admin_keytab = /var/krb5kdc/kadm5.keytab
    acl_file = /var/krb5kdc/kadm5.acl
    dict_file = /var/krb5kdc/kadm5.dict
    key_stash_file = /var/krb5kdc/.k5.FOO
    kadmin_port = 749
    max_life = 10h 0m 0s
    max_renewable_life = 7d 0h 0m 0s
    master_key_type = des-cbc-crc
    supported_enctypes = des-cbc-crc:normal des-cbc-crc:v4
  }

4. Run:

$ kdb5_util create -s

This creates the KDC principal database in /var/krb5kdc. You are prompted for the KDC master key, a password the KDC needs to operate. The key is stored in /var/krb5kdc/k5.FOO, which allows the KDC software to start without human intervention but which is obviously not wise unless the KDC machine is extremely well protected.

(The -s switch creates the stash file for the DES key)

5. Run:

$ kadmin.local
This program modifies the principal database. Issue the following kadmin commands:

kadmin.local: ktadd -k /var/krb5kdc/kadm5.keytab kadmin/admin
kadmin/local: changepw

Entry for principal kadmin/admin with kvno 4, encryption type DES cbc mode with CRC-32 added to keytab WRFILE:/var/krb5kdc/kadm5.keytab.
Entry for principal kadmin/changepw with kvno 4, encryption type DES cbc mode with CRC-32 added to keytab WRFILE:/var/krb5kdc/kadm5.keytab.

kadmin.local: add_principal -randkey host/shag.carpet.net
WARNING: no policy specified for host/shag.carpet.net@FOO; defaulting to no policy
Principal "host/shag.carpet.net@FOO" created.

kadmin.local: ktadd -k /etc/krb5.keytab host/shag.carpet.net
Entry for principal host/shag.carpet.net with kvno 3, encryption type
DES cbc mode
with CRC-32 added to keytab WRFILE:/etc/krb5.keytab.

kadmin.local: add_principal fred
WARNING: no policy specified for fred@FOO; defaulting to no policy
Enter password for principal "fred@FOO": ********
Re-enter password for principal "fred@FOO": ********
Principal "fred@FOO" created.

kadmin.local: quit

6. Now, start the KDC and the kadmin daemons, krb5kdc and kadmind.

If all goes well, you should be able to use kinit to get a TGT using the password you gave
to kadmin.local when creating the "fred" principal, klist to see the TGT, and kpasswd to
change your Kerberos password.

7. Try out kerberized SSH.

11.4.5 Kerberos-4 in OpenSSH

OpenSSH also supports Kerberos but only the older Kerberos-4 standard. The mechanics from a
user perspective are mostly the same: in a functioning Kerberos realm, you use kinit to obtain a
TGT, and then run the SSH client with KerberosAuthentication turned on (which it is by
default). The sysadmin must compile OpenSSH using --with-kerberos4, ensure there is a
Kerberos host principal with its keys installed on the SSH server machine, and turn on
KerberosAuthentication in the SSH server configuration. The host principal is
rcmd.hostname@REALM, and the keytab file is /etc/srvtab. Kerberos-Authentication is
on by default in the server only if /etc/srvtab exists when it starts.

[14] Principals in Kerberos-4 also include a name, optional instance, and realm, but are written
name.instance@REALM instead of name/instance@REALM as in Kerberos-5.

Access control for an account is via the file ~/.klogin. With Kerberos-4, it isn't necessary to
include the account's default principal in ~/.klogin if that file exists; the default principal always
has access.

Table 11-1 summarizes the salient differences between Kerberos-4 and Kerberos-5 with respect
to SSH.

| Table 11.1. Differences Between Kerberos-4 and Kerberos-5 with Respect to SSH |
|-----------------------------|-----------------------------|
| Kerberos-4                  | Kerberos-5                  |
| Host principal              | rcmd.hostname@REALM         | host/hostname@REALM         |
| Config files                | /etc/krb.conf, /etc/krb.realms | /etc/krb5.conf              |
| Server principal keys       | /etc/srvtab                | /etc/krb5.keytab            |
| Authorization file          | ~/.klogin                  | ~/.k5login                  |

11.4.5.1 Kerberos-4 compatibility mode in Kerberos-5

If you have a Kerberos-5 realm, you don't need to set up a separate Kerberos-4 KDC just to
support OpenSSH. Kerberos-5 has a version 4 (v4) compatibility mode, in which the v5 KDC
responds to v4 requests. If v4 compatibility is on, you can install v4 /etc/krb.conf and
/etc/krb.realms files that point to your existing v5 KDC, and the v4 kinit can obtain a v4 TGT.
Following the example in the previous section, these look like:
The KDC satisfies v4 requests for `rcmd.hostname@REALM` tickets, using the key of the corresponding v5 `host/hostname@REALM` principal, so you don't need to create separate "rcmd/hostname" principals in your v5 KDC. Since v4-only servers still need the principal key, you need to create a v4 version of the key file (`/etc/srvtab`) with that key; you can do this with the v5 program `kutil` to read in an existing `krb5.keytab` and write out a v4 `srvtab`. Direct cross-realm authentication also automatically works using existing cross-realm keys; however, Kerberos-4 doesn't support hierarchical realms.

Using the Kerberos-5 credentials conversion service, you can even avoid having to do a separate v4 `kinit`. On the KDC, the separate server program `krb524d` must be running. Then, after doing a v5 `kinit`, the user simply runs the program `krb524init`. This obtains a v4 TGT using the v5 one, which you can verify with the v4 `klist` command.

Note that OpenSSH and SSH1 can't interoperate using Kerberos authentication. They use the same SSH protocol messages in each case but implicitly expect encapsulated Kerberos tickets of the appropriate Kerberos version. You can't use Kerberos-5 v4 compatibility mode to overcome this limitation. We hope OpenSSH will eventually add Kerberos-5 support.

Also note that Kerberos-4 doesn't have an analog to the Kerberos-5 `kinit -A` switch. We don't know of any way to overcome the Kerberos/NAT problem using Kerberos-4. [Section 11.4.4.2] We have heard, however, that the Transarc AFS KDC ignores IP addresses in tickets, thus avoiding the problem.

### 11.4.5.2 Kerberos on Solaris

Sun Microsystems's Solaris operating system comes with a limited, special-purpose implementation of Kerberos-4, which supports Kerberos authentication for Sun's NFS and secure RPC. As far as we can tell, it doesn't suffice for compiling or running OpenSSH with Kerberos-4, so you probably want to install another Kerberos-4 package such as MIT's. Beware of confusion once you do so; the Solaris `/bin/kinit`, for instance, won't have any effect on the MIT Kerberos-4 operation.

### 11.5 Connecting Through a GatewayHost

All along we've assumed that your outgoing connectivity is unlimited: that you can establish any outgoing TCP connection you desire. Even our discussions of firewalls have assumed that they restrict only incoming traffic. In more secure (or simply more regimented) environments, this might not be the case: in fact, you might not have direct IP connectivity at all to the outside world.

In the corporate world, companies commonly require all outgoing connections to pass through a proxy server or gateway host: a machine connected to both the company network and the outside. Although connected to both networks, a gateway host doesn't act as a router, and the networks remain separated. Rather, it allows limited, application-level access between the two networks.

In this case study, we discuss issues of SSH in this environment:

- Connecting transparently to external hosts using `ssh`
• Making `scp` connections
• Running SSH-within-SSH by port forwarding

### 11.5.1 Making Transparent SSH Connections

Suppose your company has a gateway host, G, which is your only gateway to the Internet. You are logged into a client host C and want to reach a server host S outside the company network, as shown in Figure 11-14. We assume that all three machines have SSH installed.

![Figure 11.14. Proxy gateway](image)

To make a connection from client C to server S now requires two steps:

1. Connect from C to gateway G:
   ```
   # Execute on client C
   $ ssh G
   ```

2. Connect from G to server S:
   ```
   # Execute on gateway G
   $ ssh S
   ```

This works, of course, but it requires an extra manual step on the gateway, a machine you don't care about. Using agent forwarding and public-key authentication, you can avoid entering a passphrase on gateway G, but even so, the additional hop ideally should be transparent.

Worse, you can't transparently execute remote commands on server S from client C. Instead of the usual:

```
# Execute on client C
$ ssh S /bin/ls
```

you must run a remote `ssh` on gateway G that in turn contacts server S:

```
# Execute on client C
$ ssh G "ssh S /bin/ls"
```

This isn't only annoying but also can complicate automation. Imagine rewriting all your SSH-based scripts to accommodate this environment.

Fortunately, SSH configuration is flexible enough to afford a neat solution, which we now present using SSH1 features and syntax. We use public-key authentication to take advantage of the options of the `authorized_keys` file, and `ssh-agent` with agent forwarding so that authentication passes on transparently to the second SSH connection (see Figure 11-15).

![Figure 11.15. Chained SSH connections through a proxy gateway](image)
Suppose your account on gateway G is gilligan, and on server S it is skipper. First, set up your SSH client configuration file so the name S is a nickname for accessing your account on gateway G:

```bash
# ~/.ssh/config on client C
host S
    hostname G
    user gilligan
```

Next, on gateway G, associate a forced command with your chosen key to invoke an SSH connection to server S: [Section 8.2.4]

```bash
# ~/.ssh/authorized_keys on gateway G
command="ssh -l skipper S" ...
```

Now, when you invoke the command `ssh S` on client C, it connects to gateway G, runs the forced command automatically, and establishes a second SSH session to server S. And thanks to agent forwarding, authentication from G to S happens automatically, assuming you’ve loaded the appropriate key. This can be the same key you used to access gilligan@G or a different one.\[^{15}\]

\[^{15}\] Note that if you want to use this setup for an interactive connection, you need to use the `-t` option to `ssh`, to force it to allocate a tty on G. It doesn’t normally do that, because it doesn’t have any way to know that the remote command—in this case, another instance of `ssh`—needs one.

This trick not only provides a transparent connection from client C to server S, it also sidesteps the fact that the name S might not have any meaning on client C. Often in this kind of network situation, your internal network naming scheme is cut off from the outside world (e.g., split DNS with internal roots). After all, what’s the point of allowing you to name hosts you can’t reach? Thanks to the `Host` configuration keyword for SSH clients, you can create a nickname S that instructs SSH to reach that host transparently via G. [Section 7.1.3.5]

### 11.5.2 Using SCP Through a Gateway

Recall that the command:

```
$ scp ... S:file ...
```

actually runs `ssh` in a subprocess to connect to S and invoke a remote `scp` server. [Section 3.8.1] Now that we’ve gotten `ssh` working from client C to server S, you’d expect that `scp` would work between these machines with no further effort. Well, it almost does, but it wouldn’t be software if there weren’t a couple of small problems to work around:
- Problems invoking the `ssh` subprocess, due to the forced command
- Authentication difficulties due to lack of a tty

### 11.5.2.1 Passing along the remote command

The first problem is that the `ssh` command on client C sends a command to be executed on server S, that starts the `scp` server, but now that command is ignored in favor of our forced command. You have to find a way to relay the intended `scp` server command to S. To accomplish this, modify the `authorized_keys` file on gateway G, instructing `ssh` to invoke the command contained in the environment variable `SSH_ORIGINAL_COMMAND`: [Section 8.2.4.4]

```shell
# ~/.ssh/authorized_keys on gateway G
command="ssh -l skipper S $SSH_ORIGINAL_COMMAND" ...key...
```

Now the forced command invokes the proper `scp`-related command on server S. You aren't quite done, however, because this forced command unfortunately breaks our existing setup. It works fine for `ssh` invocations on client C that run a remote command (e.g., `ssh S /bin/ls`), but it fails when `ssh S` is invoked alone to run a remote shell. You see, `SSH_ORIGINAL_COMMAND` is set only if a remote command is specified, so `ssh S` dies because `SSH_ORIGINAL_COMMAND` is undefined.

You can work around this problem using the Bourne shell and its parameter substitution operator `:` as follows:

```shell
# ~/.ssh/authorized_keys on gateway G
command="sh -c 'ssh -l skipper S ${SSH_ORIGINAL_COMMAND:-}'' ...key...
```

The expression `${SSH_ORIGINAL_COMMAND:-}` returns the value of `SSH_ORIGINAL_COMMAND` if it is set, or the empty string otherwise. (In general, `${V:-D}` means "return the value of the environment variable V or the string D if V isn't set." See the `sh` manpage for more information.) This produces precisely the desired behavior, and `ssh` and `scp` commands both work properly now from client C to server S.

### 11.5.2.2 Authentication

The second `scp`-related problem is authentication for the second SSH connection, from gateway G to server S. You can't provide a password or passphrase to the second `ssh` program, since it has no tty allocated. So you need a form of authentication that doesn't require user input: either RhostsRSA, or public-key authentication with agent forwarding. RhostsRSA works as is, so if you plan to use it, you can skip to the next section. Public-key authentication, however, has a problem: `scp` runs `ssh` with the `-a` switch to disable agent forwarding. [Section 6.3.5.3] You need to reenable agent forwarding for this to work, and this is surprisingly tricky.

[16] Actually, you can hack your way around this, but it's ugly and we won't go into it.

Normally you could turn on agent forwarding in your client configuration file:

```shell
# ~/.ssh/config on client C, but this FAILS
ForwardAgent yes
```

but this doesn't help because the `-a` on the command line takes precedence. Alternatively, you might try the `-o` option of `scp`, which can pass along options to `ssh`, such as `-o ForwardAgent yes`. But in this case, `scp` places the `-a` after any `-o` options it passes where it takes precedence, so that doesn't work either.
There is a solution, though. `scp` has a `-S` option to indicate a path to the SSH client program it should use, so you create a "wrapper" script that tweaks the SSH command line as needed, and then make `scp` use it with `-S`. Place the following script in an executable file on client C, say `~/bin/ssh-wrapper`:

```bash
#!/usr/bin/perl
exec '/usr/local/bin/ssh1', map {$_ eq '-a' ? (  ) : $_} @ARGV;
```

This runs the real `ssh`, removing `-a` from the command line if it's there. Now, give your `scp` command like this:

```
scp -S ~/bin/ssh-wrapper ...
```

and it should work.

### 11.5.3 Another Approach: SSH-in-SSH(Port Forwarding)

Instead of using a forced command, here's another way to connect by SSH through a gateway: forward a port on client C to the SSH server on S, using an SSH session from C to G, and then run a second SSH session through the first (see Figure 11-16).

![Figure 11.16. Forwarded SSH connection through a proxy gateway](image)

That is:

```
# Execute on client C
$ ssh -L 2001:S:22 G

# Execute on client C in a different shell
$ ssh -p 2001 localhost
```

This connects to server S by carrying the second SSH connection (from C to S) inside a port-forwarding channel of the first (from C to G). You can make this more transparent by creating a nickname S in your client configuration file:

```
# ~/.ssh/config on client C
host S
  hostname localhost
  port 2001
```

Now the earlier commands become:
# Execute on client C
$ ssh -L2001:S:22 G

# Execute on client C in a different shell
$ ssh S

Because this technique requires a separate, manual step to establish the port forwarding, it is less transparent than the one in [Section 11.5.1]. However, it has some advantages. If you plan to use port or X forwarding between C and S with the first method, it's a little complicated. scp not only gives the -a switch to ssh to turn off agent forwarding, but also it gives -x and -o "ClearAllForwardings yes", turning off X and port forwarding. So you need to modify the earlier wrapper script to remove these unwanted options as well. [Section 11.5.2.2] Then, for port forwarding you need to set up a chain of forwarded ports that connect to one another. For example, to forward port 2017 on client C to port 143 (the IMAP port) on server S:

# ~/.ssh/config on client C
host S
  hostname G
  user gilligan

# ~/.ssh/authorized_keys on gateway G
command="ssh -L1234:localhost:143 skipper@S" ...key...

# Execute on client C
$ ssh -L2017:localhost:1234 S

This works, but it's difficult to understand, error-prone, and fragile: if you trigger the TIME_WAIT problem [Section 9.2.9.1], you have to edit files and redo the tunnel just to pick a new ephemeral port to replace 1234.

Using the SSH-in-SSH technique instead, your port and X-forwarding options operate directly between client C and server S in the usual, straightforward manner. The preceding example becomes:

# ~/.ssh/config on client C
host S
  hostname localhost
  port 2001

# Execute on client C
$ ssh -L2001:S:22 G

# Execute on client C in a different shell
$ ssh -L2017:localhost:143 S

This final command connects to server S, forwarding local port 2017 to the IMAP port on S.

11.5.4 Security Differences

The two methods just discussed differ in their security properties. Again, we assume the situation with machines C, G, and S as used earlier.

11.5.4.1 "Server-in-the-middle" attack

The first method was a chain of two SSH connections in series. The weakness with this is that if the SSH server in the middle (on G ) has been compromised, the session data is exposed. Data from C is decrypted by that server and passed to the second SSH client (also on G ), which then
reencrypts it for transmission to S. So the session plaintext is recovered on G: a compromised server there has access to it and can read and alter it at will.

The second method, with port forwarding, doesn't suffer from this weakness. The server on G is in no special position with regard to observing the forwarded SSH connection from C to S. Any attempt to read or alter that session will fail, in the same way that network snooping or an active network attack will fail.

11.5.4.2 Server authentication

On the other hand, the port forwarding method is weaker than the chain-of-connections when implemented with SSH1 or OpenSSH, because it lacks server authentication. The reason for this is that the SSH1 and OpenSSH clients both behave specially when the server address is 127.0.0.1 ("localhost"): they force acceptance of the host key, regardless of what key is actually provided. More precisely: they omit checking the host key against the known-hosts list, behaving always as if the server-provided host key were associated with "localhost" in the list.

The reason for this feature is convenience. If a user's home directory is shared between machines, the SSH client on each machine sees the same per-user known-hosts file. But the name "localhost" is special, in that on each machine it means something different: that same host. So if the user employs `ssh localhost` on multiple machines, she will constantly get spurious warnings about the host key having changed. The known-hosts file maps "localhost" to the host key of the last host on which she did this, not the current one.

So the problem here is that, since the remote IP address of the SSH session from C to S is actually localhost, it effectively omits server authentication, and is thus vulnerable to a man-in-the-middle or spoofed server attack.

SSH2 doesn't have this special treatment of localhost and so doesn't exhibit the weakness. Its known-hosts list is also more fine-grained: it maps server sockets ([host,port] pairs) to keys, rather than server hosts. This means you can have separate keys for each locally forwarded port. So, to be as secure as possible, you don't just accept the server host key the first time you use `ssh2` to connect from C to S over the forwarded port 2001 on C. Doing so circumvents server authentication for that first connection. Instead, before making the first connection, you should copy S's host key into this file on C: `~/.ssh2/hostkeys/key_2001_localhost.pub`. This associates S's host key with the socket (localhost,2001), and you will have proper server authentication for the initial forwarded connection.
Chapter 12. Troubleshooting and FAQ

SSH1, SSH2, and OpenSSH are complex products. When a problem occurs, your plan of action should be, in order:

1. Run the client and server in debug mode.
2. Consult archives of questions and answers to see if anyone else has encountered and solved this problem.
3. Seek help.

Many people jump immediately to Step 3, posting questions in public forums and waiting hours or days for a reply, when a simple `ssh -v` or FAQ can clarify the problem in moments. Be a smart and efficient technologist, and use your available resources before seeking help from the community. (Although the SSH community is eager to help if you've done your homework.)

12.1 Debug Messages: Your First Line of Defense

SSH1/SSH2 clients and servers have debugging built-in. When invoked with appropriate options, these programs emit messages about their progress and failures. You can use these messages to isolate problems.

12.1.1 Client Debugging

Most clients print debug messages when invoked with the `-v` (verbose mode) option: [Section 7.4.15]

```
$ ssh -v server.example.com
$ scp -v myfile server.example.com:otherfile
```

So many problems can be identified in verbose mode. This should be your first instinct whenever you encounter a problem.

Please take a deep breath and repeat after us:

"ssh -v is my friend..."

"ssh -v is my friend..."

"ssh -v is my friend..."

12.1.2 Server Debugging

The SSH1, SSH2, and OpenSSH servers also print debug messages when asked:

```
# SSH1, OpenSSH
$ sshd -d
# SSH2 only
$ sshd2 -v
```
In either case, the server enters a special debugging mode. It accepts a single connection, operates normally until the connection terminates, and then exits. It doesn't go into the background or create a child process to handle the connection, and it prints information on its progress to the screen (that is, to the standard error stream).

SSH2 has a more complicated system for debugging: numeric debugging levels, specified with the `-d` option, where a higher number means more information. [Section 5.8.2] In fact, `-v` for verbose mode is actually just a shorthand for `-d2`. At higher debug levels, the output is so huge that only SSH developers will likely find it of use in tracking down obscure problems. But you may need to crank up the level beyond 2 to see the information you need. For example, to have it report which algorithms are negotiated for a connection, use `-d3`. If you get the error message "TCP/IP Failure", turning up to `-d5` shows the more specific OS-level error message returned from the connection attempt.

When debugging a server, remember to avoid port conflicts with any other running SSH server. Either terminate the other server or use an alternative port number for debugging:

```
$ sshd1 -d -p 54321
```

Use the `-p` option in the client when testing this debugging instance of the server:

```
$ ssh -p 54321 localhost
```

This way, you don't interrupt or affect another `sshd` in use.

---

### The Top Ten SSH Questions

**How do I install my public key file on the remote host the first time?**

Connect by password authentication and use your terminal program's copy and paste feature. [Section 12.2.2.4]

**I put my SSH public key file mykey.pub into my remote SSH directory, but public-key authentication doesn't work.**

The public key must be referenced in your remote authorization file. [Section 12.2.2.4]

**Public-key authentication isn't working.**

Use ssh `-v`, and check your keys, files, and permissions. [Section 12.2.2.4]

**Password authentication isn't working.**

Use ssh `-v`. There are a variety of possible causes. [Section 12.2.2.2]

**Trusted-host authentication isn't working (SSH1 RhostsRSA, SSH2 hostbased).**

Use ssh `-v`. Check your four control files, hostnames, and setuid status of the SSH client program or `ssh-signer2`. [Section 12.2.2.3]

**How do I authenticate without typing a password or passphrase?**

`ssh-agent`, unencrypted keys, trusted-host authentication, or Kerberos. [Section
How do I secure FTP with port forwarding?

Forward a local port to port 21 on the FTP server for the control connection; the data connection is much harder. [Section 12.2.5.6]

X forwarding isn't working.

Don't set your remote DISPLAY variable manually. (And there are other things to check.) [Section 12.2.5.6]

Why don't wildcards or shell variables work on the scp command line?

Your local shell expands them before scp runs. Escape the special characters. [Section 12.2.5.4]

A feature of ssh or scp isn't working, but I'm sure I'm using it correctly.

Use ssh -v. Also the system configuration may be overriding your settings.

12.2 Problems and Solutions

In this section, we cover a wide range of difficulties, organized by category. Sidebar "The Top Ten SSH Questions" lists what, in our experience, are the most frequently asked of the frequently asked questions. We focus on problems that may occur in many versions of the SSH software on diverse operating systems. We don't address the following sorts of questions that rapidly become obsolete:

- Compilation problems specific to one operating system, such as "HyperLinux beta 0.98 requires the --with-woozle flag"
- Problems and bugs that are specific to one version of SSH1 or SSH2, particularly older versions

These types of problems are best solved by the SSH FAQ [Section 12.3.1] or through discussion with other SSH users.

In all questions, we will assume you have already used debug or verbose mode (e.g., ssh -v) to isolate the problem. (If you haven't, you should!)

12.2.1 General Problems

The commands ssh, scp, ssh-agent, ssh-keygen, etc., aren't doing what I expect. Even the help messages look weird.

Maybe they are SSH2 programs when you are expecting SSH1, or vice versa. Locate the executables and do an ls -l. If they are plain files, they are most likely from SSH1 or OpenSSH. If they are symbolic links, check whether they point to SSH1 or SSH2 files. (SSH2 files have names ending in "2".)

When I try to connect to an SSH server, I get the error "Connection refused."
No SSH server is running where you tried to connect. Double-check the hostname and TCP port number: perhaps the server is running on a port different from the default?

When I log in, the message of the day (/etc/motd) prints twice.

Both `sshd` and the `login` program are printing it. Disable `sshd`’s printing by setting the serverwide configuration keyword `PrintMotd` to `no`.

When I log in, I see two messages about email, such as "No mail" or "You have mail."

Both `sshd` and the `login` program are checking for mail. Prevent `sshd` from checking by setting the serverwide configuration keyword `CheckMail` to `no`.

### 12.2.2 Authentication Problems

#### 12.2.2.1 General authentication problems

**The SSH1 server says "Permission denied" and exits.**

This occurs if all authentication techniques have failed. Run your client in debug mode and read the diagnostic messages, looking for clues. Also read our solutions to specific authentication problems in the rest of this section.

**How do I authenticate without typing a password or passphrase?**

The four available authentication methods for this are:

- Public-key with `ssh-agent`
- Public-key with an unencrypted key on disk (empty passphrase)
- Trusted-host
- Kerberos (SSH1 and OpenSSH/1 only)

Automatic authentication has a number of important issues you should carefully consider before selecting from the preceding list. [Section 11.1](#)

**I get prompted for my password or passphrase, but before I have time to respond, the SSH server closes the connection.**

Your server's idle timeout value may be too short. If you are a system administrator of the server machine, set `IdleTimeout` to a larger value in the serverwide configuration file. [Section 5.4.3.3](#) If you are an end user of SSH1 or OpenSSH, set an idle-timeout value in `authorized_keys`. [Section 8.2.7](#)

**RequiredAuthentications doesn’t work.**

This feature was broken in SSH2 2.0.13, causing authentication always to fail. This problem was fixed in 2.1.0.

**SilentDeny doesn’t seem to work for any authentication method.**

`SilentDeny` has nothing to do with authentication. It applies only to access control using `AllowHosts` and `DenyHosts`. If a connection is denied access by an `AllowHosts` or `DenyHosts` value, `SilentDeny` controls whether the client sees an informative failure message or not.
12.2.2.2 Password authentication

Password authentication isn't working.

Use `ssh -v`. If the connection is being refused altogether, the SSH server is probably not running, or you are connecting to the wrong port. Port 22 is the default, but the remote system administrator might have changed it. If you see "permission denied," password authentication might be disabled in the server.

Make sure the server permits password authentication in the serverwide configuration file ("PasswordAuthentication yes" for SSH1 and OpenSSH, "AllowedAuthentications password" for SSH2). Also check your client configuration file to make sure you don't have "PasswordAuthentication no".

If you are prompted for your password, but it is rejected, you might accidentally be connecting to the wrong account. Does your local username differ from the remote username? Then you must specify the remote username when connecting:

```bash
$ ssh -l my_remote_username server.example.com
$ scp myfile my_remote_username@server.example.com:
```

If this still doesn't work, check your local client configuration file (~/.ssh/config or ~/.ssh2/ssh2_config) to make sure you haven't accidentally set the wrong value for the `User` keyword. In particular, if your configuration file contains `Host` values with wildcards, check that your current command line (the one that isn't working) isn't matching the wrong section in the file. [Section 7.1.3.4]

One common problem on the server side involves OpenSSH and Pluggable Authentication Modules configuration. PAM is a general system for performing authentication, authorization, and accounting in an application-independent fashion. If your operating system supports PAM (as Linux and HP-UX do, for example), OpenSSH will probably have been automatically compiled to use it. Unless you take the extra step of configuring PAM to support SSH, all password authentication will mysteriously fail. This is usually just a matter of copying the appropriate `sshd.pam` file from the `contrib` directory in the OpenSSH distribution, naming the copy "sshd" and placing it in the PAM configuration directory (usually `/etc/pam.d`). The `contrib` directory contains several example files for different flavors of Unix. For example, on a RedHat Linux system:

```bash
# cp contrib/redhat/sshd.pam /etc/pam.d/sshd
# chown root.root /etc/pam.d/sshd
# chmod 644 /etc/pam.d/sshd
```

If OpenSSH isn't using PAM, and password authentication still isn't working, the compilation switches `--with-md5-passwords` or `--without-shadow` might be relevant. These make no difference if PAM support is enabled in OpenSSH, because they deal with how OpenSSH reads the Unix `passwd` map. When using PAM, the OpenSSH code doesn't read the `passwd` map directly; the PAM libraries do it instead. Without PAM, though, if your system is using MD5-hashed passwords instead of the more traditional crypt (DES) hash, you must use `--with-md5-passwords`. You can tell which hash your system is using by inspecting the `/etc/passwd` and `/etc/shadow` files. The hashed password is the second field in each entry; if the password field in `/etc/passwd` is just "x", then the real entry is in `/etc/shadow` instead. MD5 hashes are much longer and contain a wider range of characters:

```bash
# /etc/shadow, MD5 hash
test:$1$tEMXcnZB$rDEZbQXJzUz4g2J4qYkRh:....
```
Finally, you can try --without-shadow if you suspect OpenSSH is trying to use the shadow password file, but your system doesn’t use it.

The server won't let me use an empty password.

Empty passwords are insecure and should be avoided. Nevertheless, you can set "PermitEmptyPasswords yes" in the serverwide configuration file. [Section 5.6.3]

12.2.2.3 Trusted-host authentication

Trusted-host authentication isn’t working (SSH1 RhostsRSA, SSH2 hostbased).

Use ssh -v. If everything looks right, check the following. Suppose the client user is orpheus@earth, and the target account is orpheus@hades—that is, on host earth, user orpheus invokes ssh hades.

For SSH1 and OpenSSH/1:

- The SSH client program must be setuid root.
- "RhostsRSAAuthentication yes" belongs in the server and client configurations.
- The client’s public host key must be in the server’s known hosts list. In this case, hades:/etc/ssh_known_hosts must contain an entry associating the name "earth" with earth’s public host key, like this:

  earth 1024 37 71641647885140363140390131934...

- The entry may be in the target account’s known hosts file instead, i.e., in hades:~orpheus/.ssh/known_hosts. Take care that "earth" is the canonical name of the client host from the server’s point of view. That is, if the SSH connection is coming from the address 192.168.10.1, then gethostbyname(192.168.10.1) on hades must return "earth", and not a nickname or alias for the host (e.g., if the hostname is river.earth.net, the lookup must not return just "river"). Note that this can involve multiple naming services, since gethostbyname can be configured to consult multiple sources to determine a translation (e.g., DNS, NIS, /etc/hosts). See /etc/nsswitch.conf. If your systems don’t agree on canonical hostnames, you’ll have no end of trouble with RhostsRSA. You can work around such problems to an extent by manually adding extra host nicknames to the known hosts file, like this:

  earth,gaia,terra 1024 37 71641647885140363140390131934...

- Edit hades:/etc/hosts.equiv or hades:~orpheus/.hosts to allow the login. Adding earth to hosts.equiv allows any nonroot user on earth to access the account by the same name on hades. Adding earth to .hosts allows orpheus@earth to access orpheus@hades.
- Some firewalls reject outbound connections from privileged ports. This prevents RhostsRSA authentication from working, since it relies on privileged source ports. You can use ssh -P to get a connection to the SSH server via a nonprivileged port, but you will have to use a different kind of authentication.

For SSH2:
- "AllowedAuthentications hostbased" in the server and client configurations.
- ssh2 doesn't need to be setuid root, but ssh-signer2 does. More precisely, it needs to be able to read the private host key, which in the normal installation means it must be setuid root.
- A copy of earth's public host key in hades:/etc/ssh2/knownhosts/earth.ssh-dss.pub (or hades:~orpheus:/ssh2/knownhosts/earth.ssh-dss.pub, if you specified "UserKnownHosts yes" on the server).
- Regarding canonical hostnames, the same comments as for RhostsRSA apply.

For OpenSSH/2:

- "DSAAuthentication yes" belongs in the server and client configurations.
- ssh must be setuid root (or otherwise able to read the client host's private host key in /etc/ssh_host_dsa_key; it doesn't require a privileged source port).
- A copy of earth's public host key in hades:/etc/ssh_known_hosts2 (or hades:~orpheus:/ssh/known_hosts2).
- The same comments as for RhostsRSA apply, regarding canonical hostnames.

### 12.2.2.4 Public-key authentication

**How do I install my public key file on the remote host the first time?**

Here's the general method:

1. Generate a key pair.
2. Copy the text of the public key into your computer's clipboard or other cut/paste buffer.
3. Log into the remote host via SSH with password authentication, which doesn't require any special files in your remote account.
4. Edit the appropriate authorization and key files on the remote host:
   - For SSH1 and OpenSSH/1, append the public key to ~/.ssh/authorized_keys.
   - For OpenSSH/2, append the public key to ~/.ssh/authorized_keys2.
   - For SSH2, paste the public key into a new .pub file in ~/.ssh2 (say, newkey.pub), and append the line "Key newkey.pub" to ~/.ssh2/authorization.
5. Log out from the remote host.
6. Log back into the remote host using public-key authentication.

When editing the remote authorization file, make sure your text editor doesn't insert line breaks into the middle of a public key. SSH1 and OpenSSH public keys are very long and must be kept on a single line.

*I put my SSH public key file mykey.pub into my remote SSH directory, but public-key authentication doesn't work.*

Placing a valid public key file (e.g., mykey.pub) in your SSH directory isn't sufficient. For SSH1 and OpenSSH/1, you must append the key (i.e., the contents of mykey.pub) to ~/.ssh/authorized_keys. For OpenSSH/2, append the key to ~/.ssh/authorized_keys2. For SSH2, you must add a line of text to ~/.ssh2/authorization, Key mykey.pub.

*Public-key authentication isn't working.*

Invoke the client in debug mode (ssh -v). Make sure:
• Your local client is using the expected identity file.
• The correct public key is on the remote host in the right location.
• Your remote home directory, SSH directory, and other SSH-related files have the correct permissions. [Section 5.4.2.1]

I'm being prompted for my login password instead of my public key passphrase. Or, my connection is rejected with the error message "No further authentication methods available." (SSH2)

There are several possible causes for both of these problems:

• Public-key authentication must be enabled in both the client and server (SSH1/OpenSSH "RSAAuthentication yes", SSH2 "AllowedAuthentications publickey").
• Specify your remote username with -l (lowercase L) if it differs from your local username, or else the SSH server will examine the wrong remote account:

  $ ssh -l jones server.example.com

• Check the file permissions in your server account. If certain files or directories have the wrong owner or careless access permissions, the SSH server refuses to perform public-key authentication. This is a security feature. Run ssh in verbose mode to reveal the problem:

  $ ssh -v server.example.com
  ...
  server.example.com: Remote: Bad file modes for /u/smith/.ssh

  In your server account, make sure that the following files and directories are owned by you and aren't world writable: ~, ~/.ssh, ~/.ssh/authorized_keys, ~/.ssh2, ~/.rhosts, and ~/.hosts.

• For SSH2, if you use the -i option to specify an identification file:

  $ ssh2 -i my-identity server.example.com

  check that my-identity is an identification file, not a private key file. (In contrast, ssh -i for SSH1 and OpenSSH expects a private key file.) Remember that SSH2 identification files are text files containing the names of private keys.

I'm being prompted for the passphrase of the wrong key.

Make sure your desired public key is in your authorization file on the SSH server machine.

Check for SSH agent problems. Are you running an agent and trying to specify another key with ssh -i or the IdentityFile keyword? The presence of an agent prevents -i and IdentityFile from working. Terminate your agent and try again.

For SSH1 and OpenSSH, if any options are specified in ~/.ssh/authorized_keys, check for typographical errors. A mistyped option causes the associated key line to be skipped silently. Remember that options are separated by commas, not whitespace.

**12.2.2.5 PGP key authentication**

After the PGP passphrase prompt, I am being prompted for my login password.
If you get prompted for your PGP key, and then your password:

```
Passphrase for pgp key "mykey": ********
smith's password:
```

and you know you're typing your passphrase correctly, then first make sure you're typing your PGP passphrase correctly. (For instance, encrypt a file with that public key and decrypt it.) If so, then there might be an incompatibility between the PGP implementations on your client and server machines. We've seen this behavior when the PGP key (generated on the client machine) doesn't have sufficient bits for the PGP implementation on the server machine. Generate a new key on the server machine.

```
I get "Invalid pgp key id number '0276C297''
```

You probably forgot the leading "0x" on the key ID, and SSH is trying to interpret a hexadecimal number as a decimal. Use `PgpKeyId 0x0276C297` instead.

### 12.2.3 Key and Agent Problems

#### 12.2.3.1 General key/agent problems

I generated a key with SSH1 and tried using it with another SSH1 client, such as NiftyTelnet SSH, F-Secure SSH Client, or SecureCRT, but the client complains that the key is in an invalid format.

First, make sure you generated the key using `ssh-keygen1`, not `ssh-keygen2`. SSH1 and SSH2 keys aren't compatible.

Next, make sure you transferred the key file using an appropriate file-transfer program. If you used FTP, confirm that the private key file was transferred in binary mode, or else the copy will contain garbage. The public key file should be transferred in ASCII mode.

```
I generated an SSH1 key and tried using it with SSH2, but it didn't work. (Or vice versa.)
```

This is normal. SSH1 and SSH2 keys aren't compatible.

I specified a key manually, using `-i` or `IdentityFile`, but it never gets used!

Are you running an agent? Then `-i` and `IdentityFile` don't have any effect. The first applicable key in the agent takes precedence.

#### 12.2.3.2 ssh-keygen

Each time I run ssh-keygen, it overwrites my default identity file.

Tell `ssh-keygen` to write its output to a different file. For `ssh-keygen` in SSH1 and OpenSSH, use the `-f` option. For `ssh-keygen2`, specify the filename as the last argument on the command line; no option is needed.

Can I change the passphrase for a key without regenerating the key?

Yes. For `ssh-keygen` in SSH1 and OpenSSH, use the `-N` option, and for `ssh-keygen2`, use the `-p` option.

How do I generate a host key?
Generate a key with an empty passphrase and install it in the correct location:

```bash
# SSH1, OpenSSH
$ ssh-keygen -N '' -b 1024 -f /etc/ssh_host_key
# SSH2 only
$ ssh-keygen2 -P -b 1024 /etc/ssh2/hostkey
```

*Generating a key takes a long time.*

Yes it may, depending on the speed of your CPU and the number of bits you have requested. DSA keys tend to take longer than RSA keys.

*How many bits should I make my keys?*

We recommend at least 1024 bits for strong security.

*What does oOo.oO.oO.oO.oO mean, as printed by ssh-keygen2?*

The manpage calls it a "progress indicator." We think it's an ASCII representation of a sine wave. Or the sound of a chattering gorilla. You can hide it with the `-q` flag.

### 12.2.3.3 ssh-agent and ssh-add

*My ssh-agent isn't terminating after I log out.*

If you use the single-shell method to start an agent, this is normal. You must terminate the agent yourself, either manually (bleah) or by including appropriate lines in your shell configuration files. [Section 6.3.2.1] If you use the subshell method, the agent automatically terminates when you log out (actually, when you exit the subshell). [Section 6.3.2.2]

*When I invoke ssh-add and type my passphrase, I get the error message "Could not open a connection to your authentication agent."

Follow this debugging process:

1. Make sure you are running an `ssh-agent` process:

   ```bash
   $ /usr/bin/ps -ef | grep ssh-agent
   smith 22719     1  0 23:34:44 ?        0:00 ssh-agent
   ```

   If not, you need to run an agent before `ssh-add` will work.

2. Check that the agent's environment variables are set:

   ```bash
   $ env | grep SSH
   SSH_AUTH_SOCK=/tmp/ssh-barrett/ssh-22719-agent
   SSH_AGENT_PID=22720
   ```

   If not, then you probably ran `ssh-agent` incorrectly, like this:

   ```bash
   # Wrong!
   $ ssh-agent
   ```
For the single-shell method, you must use `eval` with backquotes:

```bash
$ eval `ssh-agent`
```

Or for the subshell method, you must instruct `ssh-agent` to invoke a shell:

```bash
$ ssh-agent $SHELL
```

3. Make sure the agent points to a valid socket:

```bash
$ ls -lF $SSH_AUTH_SOCK
prwx——— 1 smith   0 May 14 23:37 /tmp/ssh-smith/ssh-22719-agent
```

If not, your $SSH_AUTH_SOCK variable might be pointing to an old socket from a previous invocation of `ssh-agent`, due to user error. Terminate and restart the agent properly.

### 12.2.3.4 Per-account authorization files

*My per-account server configuration isn’t taking effect.*

Check the following:

- You might be confused about which versions of SSH use which files:
  - SSH1, OpenSSH/1: `~/.ssh/authorized_keys`
  - SSH2: `~/.ssh2/authorization`
  - OpenSSH/2: `~/.ssh/authorized_keys2` (note this isn’t in `~/.ssh2`)

- Remember that the `authorized_keys` and `authorized_keys2` files contains keys, whereas the SSH2 `authorization` file contains directives referring to other key files.

- You might have a typographical error in one of these files. Check the spelling of options, and remember to separate SSH1 `authorized_keys` options with commas, not whitespace. For example:

```bash
# correct
no-x11-forwarding,no-pty 1024 35 8697511247987525784866526224505...
# INCORRECT (will silently fail)
no-x11-forwarding no-pty 1024 35 8697511247987525784866526224505...
# ALSO INCORRECT (note the extra space after "no-x11-forwarding,"
no-x11-forwarding, no-pty 1024 35 8697511247987525784866526224505...
```

12.2.4 Server Problems

#### 12.2.4.1 sshd_config, sshd2_config

*How do I get sshd to recognize a new configuration file?*

You can terminate and restart `sshd`, but there’s quicker way: send the “hangup” signal (SIGHUP) to `sshd` with `kill -HUP`.

*I changed the sshd config file and sent SIGHUP to the server. But it didn't seem to make any difference.*
sshd may have been invoked with a command-line option that overrides that keyword. Command line options remain in force and take precedence over configuration file keywords. Try terminating and restarting sshd.

12.2.5 Client Problems

12.2.5.1 General client problems

A feature of ssh or scp isn't working, but I'm sure I'm using it correctly.

The feature might have been disabled by a system administrator, either when the SSH software was compiled (Chapter 4) or during serverwide configuration (Chapter 5). Compile-time flags cannot be checked easily, but serverwide configurations are found in the files /etc/sshd_config (SSH1, OpenSSH) or /etc/ssh2/sshd2_config (SSH2). Ask your system administrator for assistance.

12.2.5.2 Client configuration file

ssh or scp is behaving unexpectedly, using features I didn't request.

The program might be responding to keywords specified in your client configuration file. [Section 7.1.3] Remember that multiple sections of the config file apply if multiple Host lines match the remote machine name you specified on the command line.

My SSH1 .ssh/config file doesn't seem to work right.

Remember that after the first use of a "Host" directive in the config file, all statements are inside some Host block, because a Host block is only terminated by the start of another Host block. The ssh1 manpage suggests that you put defaults at the end of the config file, which is correct; when looking up a directive in the config file, ssh1 uses the first match it finds, so defaults should go after any Host blocks. But don't let your own indentation or whitespace fool you. The end of your file might look like:

```
# last Host block
Host server.example.com
  User linda

# defaults
User smith
```

You intend that the username for logging into server.example.com is "linda", and the default username for hosts not explicitly listed earlier is "smith". However, the line "User smith" is still inside the "Host server.example.com" block. And since there's an earlier User statement for server.example.com, "User smith" doesn't ever match anything, and ssh appears to ignore it. The right thing to do is this:

```
# last Host block
Host server.example.com
  User linda

# defaults
Host *
  User smith
```

My .ssh2/ssh2_config file doesn't seem to work right.
See our answer to the previous question for SSH1. However, SSH2 has the opposite precedence rule: if multiple configurations match your target, then the last, not the first, prevails. Therefore your defaults go at the beginning of the file.

12.2.5.3 ssh

I want to suspend ssh with the escape sequence but I am running more than two levels of ssh (machine to machine to machine). How do I suspend an intermediate ssh?

One method is to start each ssh with a different escape character; otherwise, the earliest ssh client in the chain interprets the escape character and suspends.

Or you can be clever. Remember that if you type the escape character twice, that's the meta-escape: it allows you to send the escape character itself, circumventing its usual special function. So, if you have several chained ssh sessions all using the default escape character ~, you can suspend the n th one by pressing the Return key, then n tildes, then Control-Z.

I ran an ssh command in the background on the command line, and it suspended itself, not running unless I "fg" it.

Use the -n command-line option, which instructs ssh not to read from stdin (actually, it reopens stdin on /dev/null instead of your terminal). Otherwise, the shell's job-control facility suspends the program if it reads from stdin while in the background.

ssh prints "Compression level must be from 1 (fast) to 9 (slow, best)" and exits.

Your CompressionLevel is set to an illegal value for this host, probably in your ~/.ssh/config file. It must be an integer between 1 and 9, inclusive. [Section 7.4.11]

ssh prints "rsh not available" and exits.

Your SSH connection attempt failed, and your client was configured to fall back to an rsh connection. [Section 7.4.5.8] However, the server was compiled without rsh fallback support or with an invalid path to the rsh executable. [Section 4.1.5.12]

If you didn't expect your SSH connection to fail, run the client in debug mode and look for the reason. Otherwise, the SSH server is just not set up to receive rsh connections.

ssh1 prints "Too many identity files specified (max 100)" and exits.

SSH1 has a hardcoded limit of 100 identity files (private key files) per session. Either you ran an ssh1 command line with over 100 -i options, or your configuration file ~/.ssh/config has an entry with over 100 IdentityFile keywords. You should never see this message unless your SSH command lines and/or config files are being generated automatically by another application, and something in that application has run amok. (Or else you're doing something really funky.)

ssh1 prints "Cannot fork into background without a command to execute" and exits.

You used the -f flag of ssh1, didn't you? This tells the client to put itself into the background as soon as authentication completes, and then execute whatever remote command you requested. But, you didn't provide a remote command. You typed something like:
# This is wrong

```
$ ssh1 -f server.example.com
```

The -f flag makes sense only when you give `ssh1` a command to run after it goes into the background:

```
$ ssh1 -f server.example.com /bin/who
```

If you just want the SSH session for port-forwarding purposes, you may not want to give a command. You have to give one anyway; the SSH1 protocol requires it. Use `sleep 100000`. Don't use an infinite loop like the shell command `while true; do false; done`. This gives you the same effect, but your remote shell will eat all the spare CPU time on the remote machine, annoying the sysadmin and shortening your account's life expectancy.

**ssh1 prints “Hostname or username is longer than 255 characters” and exits.**

`ssh1` has a static limit of 255 characters for the name of a remote host or a remote account (username). You instructed `ssh1`, either on the command line or in your configuration file, to use a hostname or username that's longer than this limit.

**ssh1 prints "No host key is known for <server name> and you have requested strict checking (or ‘cannot confirm operation when running in batch mode’),” and exits.**

The client can't find the server's host key in its known-hosts list, and it is configured not to add it automatically (or is running in batch mode and so can't prompt you about adding it). You must add it manually to your per-account or systemwide known-hosts files.

**ssh1 prints "Selected cipher type ... not supported by server” and exits.**

You requested that `ssh1` use a particular encryption cipher, but the SSH1 server doesn't support it. Normally, the SSH1 client and server negotiate to determine which cipher to use, so you probably forced a particular cipher by providing the `-c` flag on the `ssh1` command line or by using the `Cipher` keyword in the configuration file. Either don't specify a cipher and let the client and server work it out, or select a different cipher.

**ssh1 prints "channel_request_remote_forwarding: too many forwards” and exits.**

`ssh1` has a static limit of 100 forwardings per session, and you've requested more.

### 12.2.5.4 scp

**scp printed an error message: "Write failed flushing stdout buffer. write stdout: Broken pipe." or "packet too long".**

Your shell startup file (e.g., `~/.cshrc`, `~/.bashrc`), which is run when `scp` connects, might be writing a message on standard output. These interfere with the communication between the two `scp1` programs (or `scp2` and `sftp-server`). If you don't see any obvious output commands, look for `stty` or `tset` commands that might be printing something.

Either remove the offending statement from the startup file or suppress it for noninteractive sessions:

```
if ($?prompt) then
  echo 'Here is the message that screws up scp.'
endif
```
The latest versions of SSH2 have a new server configuration statement, 
`AllowCshrcSourcingWithSubsystems`, which should be set to `no` to prevent this 
problem.

*scp printed an error message, "Not a regular file."*

Are you trying to copy a directory? Use the `-r` option for a recursive copy. Otherwise, you 
may be trying to copy a special file that it doesn't make sense to copy, such as a device 
node, socket, or named pipe. If you do an `ls -l` of the file in question and the first character 
in the file description is something other than `.-` (for a regular file) or `d` (for a 
directory), this is probably what's happening. You didn't really want to copy that file, did 
you?

*Why don't wildcards or shell variables work on the scp command line?*

Remember that wildcards and variables are expanded by the *local* shell first, not on the 
remote machine. This happens even before `scp` runs. So if you type:

```bash
$ scp server.example.com:a* .
```

the local shell attempts to find local files matching the pattern
`server.example.com:a*`. This is probably not what you intended. You probably 
wanted files matching `a*` on `server.example.com` to be copied to the local machine.

Some shells, notably C shell and its derivatives, simply report "No match" and exit. 
Bourne shell and its derivatives (`sh`, `ksh`, `bash`), finding no match, will actually pass the 
string `server.example.com:a*` to the server as you'd hoped.

Similarly, if you want to copy your remote mail file to the local machine, the command:

```bash
$ scp server.example.com:$MAIL .
```

might not do what you intend. `$MAIL` is expanded locally before `scp` executes. Unless 
(by coincidence) `$MAIL` is the same on the local and remote machines, the command 
won't behave as expected.

Don't rely on shell quirks and coincidences to get your work done. Instead, escape your 
wildcards and variables so the local shell won't attempt to expand them:

```bash
$ scp server.example.com:a\* .
$ scp 'server.example.com:$MAIL' .
```

See also [Appendix A](#) for specifics on `scp2`'s regular expressions.

*I used scp to copy a file from the local machine to a remote machine. It ran without errors. But when I logged into the remote machine, the file wasn’t there!*

By any chance, did you omit a colon? Suppose you want to copy the file `myfile` from the 
local machine to `server.example.com`. A correct command is:

```bash
$ scp myfile server.example.com:
```

but if you forget the final colon:

```bash
# This is wrong!
```
$ scp myfile server.example.com

`myfile` gets copied locally to a file called "server.example.com". Check for such a file on the local machine.

**How can I give somebody access to my account by scp to copy files but not give full login permissions?**

Bad idea. Even if you can limit the access to `scp`, this doesn't protect your account. Your friend could run:

```
$ scp evil_authorized_keys you@your.host:.ssh/authorized_keys
```

Oops, your friend has just replaced your `authorized_keys` file, giving himself full login permissions. Maybe you can accomplish what you want with a clever forced command, limiting the set of programs your friend may run in your account. [Section 8.2.4.3]

`scp` -p preserves file timestamps and modes. Can it preserve file ownership?

No. Ownership of remote files is determined by SSH authentication. Suppose user smith has accounts on local computer `L` and remote computer `R`. If the local smith copies a file by `scp` to the remote smith account, authenticating by SSH, then the remote file is owned by the `remote` smith. If you want the file to be owned by a different remote user, `scp` must authenticate as that different user. `scp` has no other knowledge of users and uids, and besides, only root can change file ownership (on most modern Unix variants, anyway).

**OK, scp -p doesn't preserve file ownership information. But I am the superuser, and I'm trying to copy a directory hierarchy between machines (scp -r) and the files have a variety of owners. How can I preserve the ownership information in the copies?**

Don't use `scp` for this purpose. Use `tar` and pipe it through `ssh`. From the local machine, type:

```
# tar cpf - local_dir | (ssh remote_machine "cd remote_dir; tar xpf -")
```

### 12.2.5.5 sftp2

`sftp2` reports "Cipher <name> is not supported. Connection lost."

Internally, `sftp2` invokes an `ssh2` command to contact `sftp-server`. [Section 3.8.2] It searches the user's PATH to locate the `ssh2` executable rather than a hardcoded location. If you have more than one version of SSH2 installed on your system, `sftp2` might invoke the wrong `ssh2` program. This can produce the error message shown.

For example, suppose you have both SSH2 and F-Secure SSH2 installed. SSH2 is installed in the usual place, under `/usr/local`, whereas F-Secure is installed under `/usr/local/f-secure`. You ordinarily use SSH2, so `/usr/local/bin` is in your PATH, but `/usr/local/f-secure` isn't. You decide to use the F-Secure version of `scp2` because you want the CAST-128 cipher, which SSH2 doesn't include. First, you confirm that the SSH server in question supports CAST-128:

```
$ /usr/local/f-secure/bin/ssh2 -v -c cast server
```

```debug: c_to_s: cipher cast128-cbc, mac hmac-shal, compression none
debug: s_to_c: cipher cast128-cbc, mac hmac-shal, compression none```

371
Satisfied, you try `scp2` and get this:

```
$ /usr/local/f-secure/bin/scp2 -c cast foo server:bar
FATAL: ssh2: Cipher cast is not supported.
Connection lost.
```

scp2 is running the wrong copy of `ssh2` from `/usr/local/bin/ssh2`, rather than `/usr/local/f-secure/bin/ssh2`. To fix this, simply put `/usr/local/f-secure/bin` earlier in your PATH than `/usr/local/bin`, or specify the alternative location of `ssh2` with `scp2 -S`.

The same problem can occur in other situations where SSH programs run other ones. We have run afoul of it using hostbased authentication with both 2.1.0 and 2.2.0 installed. The later `ssh2` ran the earlier `ssh-signer2` program, and the client/signer protocol had changed, causing it to hang.

`sftp2` reports "ssh_packet_wrapper_input: invalid packet received."

Although this error appears mysterious, its cause is mundane. A command in the remote account's shell startup file is printing something to standard output, even though stdout isn't a terminal in this case, and `sftp2` is trying to interpret this unexpected output as part of the SFTP packet protocol. It fails and dies.

You see, `sshd` uses the shell to start the `sftp-server` subsystem. The user's shell startup file prints something, which the SFTP client tries to interpret as an SFTP protocol packet. This fails, and the client exits with the error message; the first field in a packet is the length field, which is why it's always that message.

To fix this problem, be sure your shell startup file doesn't print anything unless it's running interactively. `tcsh`, for example, sets the variable "$interactive" if stdin is a terminal. This problem has been addressed in SSH-2.2.0 with the `AllowCshrcSourcingWithSubsystems` flag, which defaults to `no`, instructing the shell not to run the user's startup file. [Section 5.7.1]

### 12.2.5.6 Port forwarding

*I'm trying to do port forwarding, but ssh complains: "bind: Address already in use."

The port you're trying to forward is already being used by another program on the listening side (the local host if it's a `-L` forwarding or the remote host if it's a `-R` ). Try using the `netstat -a` command, available on most Unix implementations and some Windows platforms. If you see an entry for your port in the LISTEN state, you know that something else is using that port. Check to see whether you've inadvertently left another `ssh` command running that's forwarding the same port. Otherwise, just choose another, unused port to forward.

This problem can occur when there doesn't appear to be any other program using your port, especially if you've been experimenting with the forwarding feature and have repeatedly used the same `ssh` to forward the same port. If the last one of these died unexpectedly (you interrupted it, or it crashed, or the connection was forcibly closed from the other side, etc.), the local TCP socket may have been left in the TIME_WAIT state (you may see this if you used the `netstat` program as described earlier). When this happens, you have to wait a few minutes for the socket to time out of this state and become free for use again. Of course, you can just choose another port number if you're impatient.
How do I secure FTP with port forwarding?

This is a complex topic. [Section 11.2] FTP has two types of TCP connections, control and data. The control connection carries your login name, password, and FTP commands; it is on TCP port 21 and can be forwarded by the standard method. In two windows, run:

```
$ ssh -L2001:name.of.server.com:21 name.of.server.com
$ ftp localhost 2001
```

Your FTP client probably needs to run in passive mode (execute the `passive` command). FTP data connections carry the files being transferred. These connections occur on randomly selected TCP ports and can't be forwarded in general, unless you enjoy pain. If firewalls or NAT (network address translation) are involved, you may need additional steps (or it may not be possible).

X forwarding isn't working.

Use `ssh -v`, and see if the output points out an obvious problem. If not, check the following:

- Make sure you have X working before using SSH. Try running a simple X client such as `xlogo` or `xterm` first. Your local DISPLAY variable must be set, or SSH doesn't attempt X forwarding.
- X forwarding must be turned on in the client and server, and not disallowed by the target account (that is, with `no-X11-forwarding` in the `authorized_keys` file).
- `sshd` must be able to find the `xauth` program to run it on the remote side. If it can't, this should show up when running `ssh -v`. You can fix this on the server side with the `XAuthLocation` directive (SSH1, OpenSSH), or by setting a PATH (that contains `xauth`) in your remote shell startup file.
- Don't set the DISPLAY variable yourself on the remote side. `sshd` automatically sets this value correctly for the forwarding session. If you have commands in your login or shell startup files that unconditionally set DISPLAY, change the code to set it only if X forwarding isn't in use.
- OpenSSH sets the remote XAUTHORITY variable as well, placing the `xauth` credentials file under `/tmp`. Make sure you haven't overridden this setting, which should look like:

  `$ echo $XAUTHORITY
  /tmp/ssh-maPK4047/cookies`

  Some flavors of Unix actually have code in the standard shell startup files (e.g., `/etc/bashrc`, `/etc/csh.login`) that unconditionally sets XAUTHORITY to `~/.Xauthority`. If that's the problem, you must ask the sysadmin to fix it; the startup file should set XAUTHORITY only if the variable is unset.

- If you are using an SSH startup file (`/etc/sshrc` or `~/.ssh/rc`), `sshd` doesn't run `xauth` for you on the remote side to add the proxy key; one of these startup files must do it, receiving the proxy key type and data on standard input from `sshd`.

12.3 Other SSH Resources
If we haven't answered your questions in this chapter, try the following good sources of help available on the Internet.

### 12.3.1 Web Sites

The first place to turn when you run into trouble is the SSH FAQ, where many answers to common questions may be found:

http://www.employees.org/~satch/ssh/faq/

The SSH home page, maintained by SSH Communications Security, is also a good resource of general information and links to related content:

http://www.ssh.com/

as is the Secure Shell Community Site:

http://www.ssh.org/

A database of compilation errors for the SSH2 product is found at:

http://www.ssh.com/support

Information on OpenSSH can be found at:

http://www.openssh.com/

And of course, check out this book's web sites:

http://www.oreilly.com/catalog/sshtdg/
http://www.snailbook.com/

### 12.3.2 Usenet Newsgroups

On Usenet, the newsgroup *comp.security.ssh* discusses technical issues about SSH. If you don't have Usenet access, you can read and search for its articles on the Web at Deja.com:

http://www.deja.com/usenet/

or any other site that archives Usenet posts.

### 12.3.3 Mailing Lists

If you are a software developer interested in contributing to SSH or working with beta software, or if you want to discuss the installation or internals of SSH applications, consider joining the SSH mailing list. To subscribe, send an email message to *majordomo@clinet.fi* as follows:

To: majordomo@clinet.fi
Subject: (blank)
subscribe ssh
Please note that this list is for technical discussion, not for asking "Where can I find SSH for the Commodore 64?" Before subscribing, read the latest messages in the SSH mailing list archive to see if the list is appropriate for you:

http://www.cs.hut.fi/ssh-archive/

Before posting a troubleshooting question on this mailing list, run the SSH client and server in debug or verbose mode and include the full text of the debug messages in your note.

If you aren't interested in developing or testing SSH applications but want to receive major announcements about SSH, join the ssh-announce mailing list.

To: majordomo@clinet.fi
Subject: (blank)
subscribe ssh-announce

12.4 Reporting Bugs

If you believe that you've discovered a bug in an SSH implementation:

1. First check if the bug has already been reported, if there's a way to do this.
2. Report the bug to the vendor, including full details of your hardware and software configuration.

For SSH1 and SSH2, bugs should be reported by email to ssh-support@ssh.com. Additionally, SSH Communication Security has a web form for submitting SSH2 bug reports:

http://www.ssh.com/support/ssh/

For OpenSSH, bug reporting instructions are found on the web site:

http://www.openssh.com/

along with FAQs and subscription information for mailing lists.

F-Secure Corporation has support pages at:

http://www.f-secure.com/support/ssh/

and accepts bug reports at F-Secure-SSH-Support@f-secure.com.
Chapter 13. Overview of Other Implementations

SSH isn't just a Unix technology. It's been implemented also for Windows, Macintosh, Amiga, OS/2, VMS, BeOS, PalmOS, Windows CE, and Java. Some programs are original, finished products, and others are ports of SSH1 or SSH2 undertaken by volunteers and in various stages of completion.

For the remainder of this book, we cover several robust implementations of SSH for Windows (95, 98, NT, 2000) and the Macintosh. These are complete, usable products, in our opinions. We also provide pointers to other implementations if you wish to experiment with them.

We have set up a web page pointing to all SSH-related products that we know. From this book's catalog page:

http://www.oreilly.com/catalog/sshtdg/

follow the link labeled Authors' Web Site, or visit us directly at:

http://www.snailbook.com/

Also check out this third-party page documenting free SSH implementations:

http://www.freessh.org/

13.1 Common Features

Every SSH implementation has a different set of features, but they all have one thing in common: a client program for logging into remote systems securely. Some clients are command-line based, and others operate like graphical terminal emulators, opening windows with dozens of configurable settings.

The remaining features vary widely across implementations. Secure file copy (scp and sftp), remote batch command execution, SSH servers, SSH agents, and particular authentication and encryption algorithms are found in only some of the products.

Nearly all implementations include a generator of public and private keys. For example, ports of SSH1/SSH2 have ssh-keygen, F-Secure SSH Client has Keygen Wizard, and SecureCRT has Key Generation Wizard. NiftyTelnet SSH for the Macintosh is a notable exception: it can't generate keys, but it accepts keys generated by other programs in the standard SSH-1 format.

13.2 Covered Products

For Windows, we cover:

- F-Secure SSH Client, a commercial SSH client by F-Secure Corporation, supporting SSH-1 and SSH-2 (a Macintosh version is also available)
- SecureCRT, a commercial SSH client by Van Dyke Technologies, supporting SSH-1 and SSH-2
- A Windows port of SSH1 by Sergey Okhapkin

For the Macintosh, we cover:

- NiftyTelnet SSH, a free SSH client by Jonas Walldén, implemented on top of the freeware Telnet client, NiftyTelnet

### 13.3 Table of Products

Unfortunately we can't cover every SSH implementation, but here are summaries to aid your explorations. The following tables list the major features of every SSH implementation we have encountered, sorted by platform, excluding the Unix products discussed in the previous part of the book (SSH1, SSH2, OpenSSH, F-Secure SSH). The meanings of the entries are described in this first table:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>The product name. If followed by &quot;(recommended),&quot; we have evaluated the program and recommend its use. If a product isn't listed as recommended, it might still be good, but we didn't evaluate it thoroughly.</td>
</tr>
<tr>
<td>Platform</td>
<td>Does it run on Windows, Macintosh, Unix, etc.? We don't list specific Windows variants (NT, 98, 2000, etc.) because we couldn't test them all. Contact the vendor for details.</td>
</tr>
<tr>
<td>Version</td>
<td>What is the most recent version number at press time?</td>
</tr>
<tr>
<td>License or distribution</td>
<td>How may this program be distributed? We provide only a summary of the licensing information; see the product documentation for full information.</td>
</tr>
<tr>
<td>Protocol</td>
<td>Does it implement SSH-1, SSH-2, or both?</td>
</tr>
<tr>
<td>Remote logins</td>
<td>Can the product open a login shell to a remote machine? We write either &quot;ssh&quot; to denote a command-line interface à la SSH1 or SSH2, or &quot;terminal program&quot; to denote a graphical interface</td>
</tr>
<tr>
<td>Remote commands</td>
<td>Can it invoke individual commands on a remote SSH server machine, in the manner of the ssh client (i.e., providing a command string as a final argument)?</td>
</tr>
<tr>
<td>File transfer</td>
<td>What program, if any, transmits files securely between machines?</td>
</tr>
<tr>
<td>Server</td>
<td>Does it include an SSH server?</td>
</tr>
<tr>
<td>Authentication</td>
<td>What forms of authentication are supported?</td>
</tr>
<tr>
<td>Key generation</td>
<td>Can it generate private/public key pairs?</td>
</tr>
<tr>
<td>Agent</td>
<td>Does it include an SSH agent?</td>
</tr>
<tr>
<td>Forwarding</td>
<td>Does it support port forwarding, X forwarding, both, or neither?</td>
</tr>
<tr>
<td>Notes</td>
<td>General information and supporting details.</td>
</tr>
<tr>
<td>Contact</td>
<td>URL to locate the software.</td>
</tr>
</tbody>
</table>

The remainder of this section is an extended table summarizing the many SSH implementations.
<table>
<thead>
<tr>
<th>Protocol</th>
<th>SSH-1</th>
<th>SSH-1</th>
<th>SSH-1</th>
<th>SSH-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remote logins</td>
<td>Terminal program</td>
<td>ssh</td>
<td>Terminal program</td>
<td>Terminal program</td>
</tr>
<tr>
<td>Remote commands</td>
<td>No</td>
<td>ssh</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>File transfer</td>
<td>No</td>
<td>scp</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Server</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Authentication</td>
<td>Password, public-key</td>
<td>Password, public-key, trusted-host</td>
<td>Password, public-key</td>
<td>Password</td>
</tr>
<tr>
<td>Key generation</td>
<td>ssh-keygen</td>
<td>ssh-keygen</td>
<td>?</td>
<td>No</td>
</tr>
<tr>
<td>Agent</td>
<td>No</td>
<td>?</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Forwarding</td>
<td>No</td>
<td>Port, X</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Notes</td>
<td>Integration of NapsaTerm with SSH1 1.2.26; requires 68020 or greater CPU</td>
<td>Port of SSH1 1.2.26, Requires Java AWT 1.1</td>
<td>Part of a Java Telnet applet</td>
<td></td>
</tr>
</tbody>
</table>

Contact


<table>
<thead>
<tr>
<th>Name</th>
<th>MindTerm (recommended)</th>
<th>BetterTelnet</th>
<th>F-Secure SSH Client</th>
<th>NiftyTelnet SSH (recommended)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform</td>
<td>Java</td>
<td>Macintosh</td>
<td>Macintosh</td>
<td>Macintosh</td>
</tr>
<tr>
<td>Version</td>
<td>1.1</td>
<td>2.0fc1</td>
<td>2.1</td>
<td>1.1 R3</td>
</tr>
<tr>
<td>License or distribution</td>
<td>GNU Public License</td>
<td>GNU Public License</td>
<td>Commercial</td>
<td>Freeware</td>
</tr>
<tr>
<td>Protocol</td>
<td>SSH-1</td>
<td>See Notes</td>
<td>SSH-1, SSH-2</td>
<td>SSH-1</td>
</tr>
<tr>
<td>Remote logins</td>
<td>Terminal program</td>
<td>See Notes</td>
<td>Terminal program</td>
<td>Terminal program</td>
</tr>
<tr>
<td>Remote commands</td>
<td>Yes</td>
<td>See Notes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>File transfer</td>
<td>scp, tunneled ftp</td>
<td>See Notes</td>
<td>Tunneled ftp</td>
<td>Graphical scp</td>
</tr>
<tr>
<td>Server</td>
<td>No</td>
<td>See Notes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Authentication</td>
<td>Password, public-key, trusted-host, TIS, sdtoken</td>
<td>See Notes</td>
<td>Password, public-key</td>
<td>public-key</td>
</tr>
<tr>
<td>Key generation</td>
<td>Yes</td>
<td>See Notes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Agent</td>
<td>No</td>
<td>See Notes</td>
<td>No</td>
<td>No (but can remember your passphrase)</td>
</tr>
<tr>
<td>Forwarding</td>
<td>Port, X</td>
<td>See Notes</td>
<td>Port, X</td>
<td>No</td>
</tr>
<tr>
<td>Notes</td>
<td>Can work as a standalone program or</td>
<td>SSH support is absent at press time (due to Windows version also</td>
<td>Minimal but useful</td>
<td></td>
</tr>
</tbody>
</table>

378
an applet; tested on many operating systems

former export restrictions) but is due back soon

available

Contact

<table>
<thead>
<tr>
<th>Name</th>
<th>SSHDOS</th>
<th>SSHOS2</th>
<th>Top Gun SSH</th>
<th>lsh</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform</td>
<td>MS-DOS</td>
<td>OS/2</td>
<td>PalmOS</td>
<td>Unix</td>
</tr>
<tr>
<td>Version</td>
<td>0.4</td>
<td>v03</td>
<td>1.2</td>
<td>1.0.3</td>
</tr>
<tr>
<td>License or distribution</td>
<td>GNU Public License</td>
<td>?</td>
<td>Freely distributable</td>
<td>GNU Public License</td>
</tr>
<tr>
<td>Protocol</td>
<td>SSH-1</td>
<td>SSH-1</td>
<td>SSH-1</td>
<td>SSH-2</td>
</tr>
<tr>
<td>Remote logins</td>
<td>Yes</td>
<td>ssh, terminal program</td>
<td>Terminal program</td>
<td>Yes</td>
</tr>
<tr>
<td>Remote commands</td>
<td>No</td>
<td>ssh</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>File transfer</td>
<td>No</td>
<td>scp</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Server</td>
<td>No</td>
<td>Unfinished</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Authentication</td>
<td>Password</td>
<td>Password, public-key, trusted-host</td>
<td>Password</td>
<td>Password, public-key, SRP</td>
</tr>
<tr>
<td>Key generation</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Agent</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Forwarding</td>
<td>No</td>
<td>Port, X</td>
<td>No</td>
<td>Port</td>
</tr>
<tr>
<td>Notes</td>
<td>Minimal; runs on low-end machines; based on PuTTY and SSH1 1.2.27</td>
<td>Based on SSH1 1.2.13</td>
<td>Based on Top Gun Telnet for the Palm Pilot</td>
<td>A promising work in progress, but not secure yet</td>
</tr>
</tbody>
</table>

Contact

[5] [http://www.appgate.org/products/mindterm]

[6] [http://www.cstone.net/~rbraun/mac/telnet/]

[7] [http://www.f-secure.com/]

[8] [http://www.lysator.liu.se/~jonasw/freeware/niftyssh/]

<table>
<thead>
<tr>
<th>Name</th>
<th>ossh</th>
<th>FISH</th>
<th>sshexec.com</th>
<th>AppGate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform</td>
<td>Unix</td>
<td>VMS</td>
<td>VMS</td>
<td>Windows, Unix, Macintosh</td>
</tr>
<tr>
<td>Version</td>
<td>1.5.6</td>
<td>0.6-1</td>
<td>5alpha1</td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>-------</td>
<td>-------</td>
<td>---------</td>
<td></td>
</tr>
<tr>
<td>License or distribution</td>
<td>BSD License</td>
<td>Freely distributable</td>
<td>Freeware</td>
<td>Commercial</td>
</tr>
<tr>
<td>Protocol</td>
<td>SSH-1</td>
<td>SSH-1</td>
<td>SSH-1</td>
<td></td>
</tr>
<tr>
<td>Remote logins</td>
<td>ssh</td>
<td>Yes</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>Remote commands</td>
<td>ssh</td>
<td>Yes</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>File transfer</td>
<td>scp</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Server</td>
<td>ssdh</td>
<td>No</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Authentication</td>
<td>Password, public-key, trusted-host</td>
<td>Password, public-key, trusted-host, TIS (untested)</td>
<td>Password public-key</td>
<td></td>
</tr>
<tr>
<td>Key generation</td>
<td>ssh-keygen</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Agent</td>
<td>ssh-agent</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Forwarding</td>
<td>Port, X</td>
<td>No</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Notes</td>
<td>Port of SSH1 1.2.12</td>
<td>A VMS server: a work in progress; not for novices</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Contact

[14] [http://www.free.lp.se/fish/]
[15] [http://www.er6.eng.ohio-state.edu/~jonesd/ssh/]
[16] [http://www.appgate.com/]

<table>
<thead>
<tr>
<th>Name</th>
<th>Chaffee Port</th>
<th>Free FiSSH</th>
<th>F-Secure SSH Client (recommended)</th>
<th>Mathur Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform</td>
<td>Windows</td>
<td>Windows NT, 2000</td>
<td>Windows</td>
<td>Windows</td>
</tr>
<tr>
<td>Version</td>
<td>1.2.14a</td>
<td>?</td>
<td>4.1</td>
<td>1.2.22-Win32-beta1</td>
</tr>
<tr>
<td>License or distribution</td>
<td>?</td>
<td>Free for noncommercial use</td>
<td>Commercial</td>
<td>Some GNU Public License, some other</td>
</tr>
<tr>
<td>Protocol</td>
<td>SSH-1</td>
<td>SSH-1</td>
<td>SSH-1, SSH-2</td>
<td>SSH-1</td>
</tr>
<tr>
<td>Remote logins</td>
<td>ssh</td>
<td>Terminal program</td>
<td>Terminal program and ssh2 command-line client</td>
<td>ssh</td>
</tr>
<tr>
<td>Remote commands</td>
<td>ssh</td>
<td>?</td>
<td>ssh2</td>
<td>ssh</td>
</tr>
<tr>
<td>File transfer</td>
<td>scp</td>
<td>?</td>
<td>scp2, sftp2, graphical sftp client</td>
<td>scp</td>
</tr>
<tr>
<td>Server</td>
<td>No</td>
<td>?</td>
<td>No</td>
<td>ssdh</td>
</tr>
<tr>
<td>Authentication</td>
<td>Password, public-key</td>
<td>?</td>
<td>Password, public-key</td>
<td>Password, public-key</td>
</tr>
</tbody>
</table>
### Name

<table>
<thead>
<tr>
<th>Key generation</th>
<th>Metro State SSH (MSSH)</th>
<th>Okhapkin Port</th>
<th>PenguNet</th>
<th>PuTTY (recommended)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Forwarding</td>
<td>Port, X</td>
<td>Port, X</td>
<td>X, port</td>
<td></td>
</tr>
<tr>
<td>Notes</td>
<td>Undocumented; based on SSH1 1.2.14</td>
<td>Unstable, in our experience (hence much missing information in this entry)</td>
<td>Also available for Macintosh</td>
<td>Barely documented alpha software, from 1998; port of SSH1 1.2.22 with cygus dll.</td>
</tr>
</tbody>
</table>

### Contact


<table>
<thead>
<tr>
<th>Name</th>
<th>SSH Secure Shell (recommended)</th>
<th>SecureCRT (recommended)</th>
<th>SecureFX (recommended)</th>
<th>SecureKoalaTerm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform</td>
<td>Windows</td>
<td>Windows</td>
<td>Windows</td>
<td>Windows</td>
</tr>
<tr>
<td>Version</td>
<td>2.1.0</td>
<td>3.1.2</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>License or distribution</td>
<td>Free for noncommercial use</td>
<td>Commercial</td>
<td>Commercial</td>
<td>Shareware</td>
</tr>
<tr>
<td>Protocol</td>
<td>SSH-2</td>
<td>SSH-1, SSH-2</td>
<td>SSH-2</td>
<td>SSH-1, SSH-2</td>
</tr>
<tr>
<td>Remote logins</td>
<td>Terminal program</td>
<td>Terminal program</td>
<td>No</td>
<td>Terminal emulator</td>
</tr>
<tr>
<td>Remote commands</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>File transfer</td>
<td>Graphical scp2</td>
<td>Zmodem (secure)</td>
<td>FTP (secure)</td>
<td>Zmodem (secure)</td>
</tr>
<tr>
<td>Server</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Authentication</td>
<td>Password, public-key</td>
<td>Password, public-key, TIS</td>
<td>Password, public-key</td>
<td>Password, public-key</td>
</tr>
<tr>
<td>Key generation</td>
<td>Yes</td>
<td>RSA, DSA</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Agent</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Forwarding</td>
<td>Port, X</td>
<td>Port, X</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Notes</td>
<td>This recent product could bring SSH2 to the masses; the scp2 client is particularly nice, emulating the Windows Explorer, permitting remote files to be dragged securely between machines; extensive documentation; SSH2 server is a separate product</td>
<td>A solid performer; our favorite of the commercial Windows clients</td>
<td>A secure, graphical FTP client for SSH2</td>
<td>Graphical terminal emulator with SSH support</td>
</tr>
</tbody>
</table>

**Contact**


[26] [http://www.vandyke.com/](http://www.vandyke.com/)

[27] [http://www.vandyke.com/](http://www.vandyke.com/)

[28] [http://www.midasoft.com/](http://www.midasoft.com/)

<table>
<thead>
<tr>
<th>Name</th>
<th>therapy Port</th>
<th>TTSSH (recommended)</th>
<th>Zoc</th>
<th>sshCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform</td>
<td>Windows</td>
<td>Windows</td>
<td>Windows</td>
<td>Windows CE</td>
</tr>
<tr>
<td>Version</td>
<td>0.2</td>
<td>1.5.1</td>
<td>3.14</td>
<td>1.00.40</td>
</tr>
<tr>
<td>License or distribution</td>
<td>See Notes</td>
<td>Freely distributable</td>
<td>Commercial</td>
<td>Freeware</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------</td>
<td>----------------------</td>
<td>------------</td>
<td>----------</td>
</tr>
<tr>
<td>Protocol</td>
<td>See Notes</td>
<td>SSH-1</td>
<td>SSH-1</td>
<td>SSH-1</td>
</tr>
<tr>
<td>Remote logins</td>
<td>See Notes</td>
<td>Terminal program</td>
<td>Terminal program</td>
<td>Terminal program</td>
</tr>
<tr>
<td>Remote commands</td>
<td>See Notes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>File transfer</td>
<td>See Notes</td>
<td>Kermit, Xmodem, Zmodem, B-Plus, Quick-VAN (all secure)</td>
<td>Kermit, Ymodem, Zmodem</td>
<td>No</td>
</tr>
<tr>
<td>Server</td>
<td>See Notes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Authentication</td>
<td>See Notes</td>
<td>Password, public-key, trusted-host, TIS</td>
<td>Password</td>
<td>Password</td>
</tr>
<tr>
<td>Key generation</td>
<td>See Notes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Agent</td>
<td>See Notes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Forwarding</td>
<td>See Notes</td>
<td>Port, X</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Notes</td>
<td>Unsupported and no longer in development; based on SSH1 1.2.20</td>
<td>Popular; an SSH extension to Teraterm Pro, a free terminal program</td>
<td>Full-featured terminal program</td>
<td>Currently this is beta software</td>
</tr>
</tbody>
</table>

Notes

[29] [http://guardian.htu.tuwien.ac.at/therapy/ssh/](http://guardian.htu.tuwien.ac.at/therapy/ssh/)
[32] [http://www.movsoftware.com/sshce.htm](http://www.movsoftware.com/sshce.htm)

### 13.4 Other SSH-Related Products

*SecPanel* is a graphical, point-and-click manager for SSH client connections; it's written in the programming language *tcl*:

- [http://www2.wiwi.uni-marburg.de/~leich/soft/secpanel/](http://www2.wiwi.uni-marburg.de/~leich/soft/secpanel/)

*ssh.el* is an Emacs interface for *ssh* client connections:

- [http://munitions.vipul.net/software/network/ssh/ssh.el](http://munitions.vipul.net/software/network/ssh/ssh.el)


*ssh-keyscan* is a replacement for *ssh-make-known-hosts*, purportedly much faster: [Section 4.1.6](#)

- [ftp://cag.lcs.mit.edu/pub/dm/source/ssh-keyscan-0.3.tar.gz](ftp://cag.lcs.mit.edu/pub/dm/source/ssh-keyscan-0.3.tar.gz)
Chapter 14. SSH1 Port by Sergey Okhapkin (Windows)

Numerous programmers have attempted to port SSH1 to Windows. Most ports that we've seen are unfinished, no longer in development, or distributed without source code. The best ports we've found are by Sergey Okhapkin, so we cover his work in this chapter. We'll call the software Okhapkin's SSH1 to distinguish it from SSH1.

Okhapkin's software works fine, but installation is difficult. For this reason we recommend it for advanced Windows users only. Ideally you should be familiar with MS-DOS environment variables, bzip2 compressed files, tar archives, the Windows NT Resource Kit, and most of all, installing applications manually on your PC. If these are alien concepts, consider a different SSH program for Windows. On the other hand, if you persevere through the installation, you get a powerful, command line-based SSH for free.

Okhapkin has done separate ports of SSH1 Versions 1.2.26 and 1.2.27, and SSH2 Version 2.0.13. We cover the 1.2.26 port since we had the least trouble installing it.

14.1 Obtaining and Installing Clients

Okhapkin's SSH1 is found on the author's web site in Russia:

http://miracle.geol.msu.ru/sos/

The software is distributed in a format that is probably unfamiliar to most Windows users. First, the software has been packed into a tar archive, which is a common file format on Unix systems. Then the archive has been compressed with bzip2, a compression utility popular among Linux users. For example, the "bzipped" tar archive for Okhapkin's Version 1.2.26 port is ssh-1.2.26-cygwinb20.tar.bz2.

In this version of Okhapkin's SSH1, the clients (ssh1, scp1) run under 32-bit Windows systems; we installed them under Windows 95. The server (sshd) runs only on Windows NT.

For the conservative installation we describe, you need 40 MB of disk space to hold both SSH and the Cygwin support software and another 20 MB during installation, so make sure to have 60 MB free. SSH itself requires only 1 MB, so if you want to save space after the installation, you can delete most of Cygwin.

14.1.1 Prepare Folders

Before you start installing software, create the following folders on your C: drive:

C:\usr
C:\usr\local
C:\usr\local\bin
C:\etc
C:\home
C:\home\.ssh  Note the period!
C:\tmp
To create C:\home\.ssh you must use the DOS mkdir command. Windows doesn't create folders with names beginning with a period.

C:\> mkdir C:\home\.ssh

14.1.2 Prepare autoexec.bat

You need to make two changes to your autoexec.bat file. First, add the folder C:\usr\local\bin to your MS-DOS search path. This is done by appending the following line to the file:

PATH=%PATH%;C:\usr\local\bin;C:\Cygwin\bin

Next, set the environment variable CYGWIN to have the value "tty":

SET CYGWIN=tty

This is required so the ssh1 client can run interactively. Finally, save autoexec.bat, open an MS-DOS command line, and apply your changes:

C:\> C:\autoexec

14.1.3 Create a Password File

On Unix, the file /etc/passwd contains login names, passwords, and other information about users. You must create a similar file on the PC to satisfy Okhapkin's SSH1 clients, because they need a login name to operate.

In the folder C:\etc you created earlier, create a one-line file called passwd. The line has seven fields, separated by colons:

1. A login name of your choice, which can be a string of letters and digits.
2. An asterisk.
3. An integer greater than 0.
4. An integer greater than 0.
5. Your full name.
6. The folder /home, where your SSH folder is created. Note the direction of the slash; it's not the MS-DOS folder separator, but the slash on the question-mark key.
7. The program /command.com. Again, note the slash.

This is the format of a passwd entry on Unix. For Okhapkin's SSH1, only fields 1 and 6 have any effect. The rest contain reasonable values should they ever be needed. Here's an example:

smith:*:500:50:Amy Smith:/home:/command.com

14.1.4 Install Cygwin

Cygwin is a wonderful collection of command-line programs. They are ports of GNU software ([http://www.gnu.org](http://www.gnu.org)) that run on Windows thanks to a library of code, the Cygwin DLL known as cygwin1.dll. Okhapkin's SSH1 requires this DLL, so after you install Cygwin, you may delete most of the other files. However, the whole Cygwin distribution is so useful we hope you'll keep it. The software is available from:

Install the binary release: the source code is unneeded for our purposes. The official download and installation can take quite some time, so you might consider downloading only cygwin1.dll and not the many accompanying programs. At press time, it is located on the Cygwin mirror machines (reachable from the URL above) in the /pub/cygwin/latest/cygwin folder. The distribution is in gzipped tar format (.tar.gz filename suffix), which WinZip for Windows can unpack. Copy cygwin1.dll to the folder C:usr\local\bin you created earlier.

14.1.5 Install bzip2

*bzip2* is a program for compressing and uncompressing files. A Windows version is available from:

http://sourceware.cygnus.com/bzip2/

Download the program to the folder C:usr\local\bin. The program is ready to run without any installation. Its name at press time is bzzip2095d_win32.exe, but this could change as future revisions are released.

Rename the *bzzip2* executable to *bzzip2.exe*:

```
C:\> cd \usr\local\bin
C:\usr\local\bin> rename bzzip2095d_win32.exe bzzip2.exe
```

14.1.6 Install Okhapkin’s SSH1

Download Okhapkin’s SSH1 Version 1.2.26 from:

http://miracle.geol.msu.ru/sos/

The filename is ssh-1.2.26-cygwinb20.tar.bz2. Because the name has multiple periods, your download software might automatically rename the file, eliminating all periods but the last, e.g., ssh-1_2_26-cygwinb20.tar.bz2.

Uncompress the file with *bzzip2* to produce a *tar* file:

```
C:\temp> bzzip2 -d ssh-1_2_26-cygwinb20.tar.bz2
```

Extract the tar file in the root of the C: drive. This unpacks files into C:usr:

```
C:\temp> cd \
C:\> tar xvf \temp\ssh-1_2_26-cygwinb20_tar
```

If you skipped installing the full Cygwin package, [Section 14.1.4] you might not have a *tar* program. The popular WinZip program for Windows is also capable of unpacking the *tar* file (after you run *bzzip2*). Be sure to unpack it into the root of the C: drive.

The SSH1 client software is now installed.

14.1.7 Create a Key Pair

Before running Okhapkin’s SSH1 clients, set up your SSH folder and generate a key pair for public-key authentication. This is done by running *ssh-keygen1*:

```
C:\> ssh-keygen1
```

386
In the C:\home\.ssh folder, \textit{ssh-keygen1} creates a private key file \textit{identity} and public key file \textit{identity.pub}. The output looks something like the following. Ignore the line \textit{w: not found} caused by a harmless difference between Unix and Windows.

\begin{verbatim}
Initializing random number generator...
w: not found
Generating p: ................++ (distance 352)
Generating q: ............++ (distance 140)
Computing the keys...
Testing the keys...
Key generation complete.
\end{verbatim}

\textit{ssh-keygen1} then prompts for a file in which to save the key. Accept the default by pressing the Enter key:

\begin{verbatim}
Enter file in which to save the key (/home/.ssh/identity): [press Enter]
\end{verbatim}

You are then prompted for a passphrase for your private key. Choose a good one and type it twice. It doesn't display onscreen.

\begin{verbatim}
Enter passphrase: ********
Enter the same passphrase again: ********
\end{verbatim}

Your key pair is now generated and saved in the folder C:\home\.ssh. Copy your public key (\textit{identity.pub}) to any SSH server machine where you want to connect, appending it to your remote \texttt{~/.ssh/authorized_keys} file. [Section 2.4.3]

\subsection{14.1.8 Log into a Remote Host with ssh1}

You are ready to connect! Run the \textit{ssh1} client, providing your remote login name. Suppose it is "smith" on the SSH server machine \texttt{server.example.com}:

C:\> ssh1 -l smith server.example.com

On your first attempt, \textit{ssh1} adds the remote host to its known hosts database. [Section 2.3.1] Answer yes and continue:

\begin{verbatim}
Host key not found from the list of known hosts.
Are you sure you want to continue connecting (yes/no)? yes
Host 'relativity.cs.umass.edu' added to the list of known hosts.
\end{verbatim}

Finally, you're prompted for your passphrase:

\begin{verbatim}
Enter passphrase for RSA key 'You@YourPC': ********
\end{verbatim}

If all goes well, you are now logged into the remote host via SSH. You can also run individual commands by SSH in the usual way, providing a command at the end of the line:

C:\> ssh1 -l smith server.example.com /bin/who

\subsection{14.1.9 Copy Files Securely with scp1}

Secure copying should also be possible with \textit{scp1}. Try copying a file to the remote machine:
14.2 Client Use

Okhapkin's SSH1 supports most of the SSH1 features found in Chapter 5 through Chapter 7 and Chapter 9. Just substitute C:\home\.ssh wherever you see ~/.ssh. For example, you can create a client configuration file in C:\home\.ssh\config. [Section 7.1.3]

Unfortunately, the SSH agent doesn't run under Windows, so none of the information in Chapter 6 about ssh-agent and ssh-add applies.

14.3 Obtaining and Installing the Server

Okhapkin's SSH1 server, sshd, can run under Windows NT installed as a service. It supports most server configuration features from Chapter 5 with the notable exception of public-key authentication. NT login authentication requires an NT username and password, and SSH can't get around this barrier to provide authentication by public key.

Like the clients, the server requires tricky installation. We'll assume you have already installed the Cygwin library and the SSH1 clients.

14.3.1 Obtain sshd

Sergey Okhapkin makes sshd 1.2.26 available on his site in two forms: as a precompiled executable or as source code diffs. We used the executable. Additionally, some other folks have created packages containing Sergey's executable and other support files. One of our favorites is:

http://www.gnac.com/techinfo/ssh_on_nt/

14.3.2 Obtain the NT Resource Kit

To run sshd as an NT service, you need three programs from the NT Resource Kit: instsrv.exe, srvany.exe, and kill.exe. The first two are utilities for turning ordinary programs into NT services. The third is for killing processes that can't be killed by the NT Task Manager.

14.3.3 Create an Administrative User

sshd will be invoked as an NT service running under an administrative user's account, so now let's create that administrative user. Run User Manager and do the following:

1. Create a local user called (say) root.
2. Make root a member of the Administrators group.
3. Under "Options/User Rights", check the checkbox "Show Advanced User Rights".

Now grant root the following rights:

- Act as part of the operating system
- Increase quotas
- Log on as a service
• Replace a process level token

Close the application, and you're ready to continue.

### 14.3.4 Install the Server

First copy the server program, sshd.exe, to a folder of your choice, say, C:\Bin. To complete the installation, you must convert *sshd* to an NT service run by your new administrative user and create some registry entries:

1. To install the server as an NT service, run the following command, assuming your administrative user is root, your NT Resource Kit programs are in C:\reskit, and your computer's name is *mypc*. (This is one command on a single line.)

   ```
   C:\> C:\reskit\instsrv.exe SecureShellDaemon
   C:\reskit\svrany.exe -a mypc\root -p root
   ```

2. Create the following registry entries. HKLM means HKEY_LOCAL_MACHINE:
   - In HKLM\SYSTEM\CurrentControlSet\Services\SecureShellDaemon, create a string value called "ObjectName" with the value "LocalSystem".
   - In HKLM\SYSTEM\CurrentControlSet\Services\SecureShellDaemon\Parameters, create a string value "Application" with the value "C:\Bin\sshd.exe" , and a string value "AppParameters" with value "-f /etc/sshd_config".

### 14.3.5 Generate Host Key

Your server needs a host key to identify it uniquely to SSH clients. [Section 5.4.1.1] Use the *ssh-keygen1* program to generate it and store the key pair in C:\etc:

   ```
   C:\> ssh-keygen1 -f /etc/ssh_host_key -N "" -C ""
   ```

### 14.3.6 Edit sshd_config

Your server is almost ready to run. Now it's time to create a server-wide configuration file so *sshd* behaves according to your system's security policy. [Section 5.3.1] On NT this file resides in C:\etc\sshd_config. For our recommended settings, see Chapter 10.

Be sure to indicate the correct locations of files, such as the host key. In Cygwin, "/" stands for the root of your boot drive. For example:

   ```
   HostKey /etc/ssh_host_key
   PidFile /etc/sshd.pid
   RandomSeed /etc/ssh_random_seed
   ```

If you make changes in sshd_config while the SSH server is running, you must terminate and restart *sshd* for those changes to take effect. [Section 14.3.9] Stopping and restarting the service with the Services control panel isn't sufficient.

### 14.3.7 Run the Server

To run *sshd*, open the Services control panel and look for the service SecureShellDaemon. Select it, and click the Start button. That's it! In the NT Task Manager, the process shows up as sshd.exe.
14.3.8 Test the Server

If you've installed both *sshd* and *ssh1* on your local PC, try connecting to yourself:

C:\> ssh1 localhost
smith@127.0.0.1's password: ********

Otherwise, try connecting from another site:

$ ssh1 -l smith mypc.mydomain.org
smith@mypc.mydomain.org's password: ********

If the connection doesn't work, use *ssh1 -v* to print diagnostic output and figure out the problem.

14.3.9 Terminate the Server

To terminate an NT service, one normally uses the Stop button on the Services control panel. Unfortunately, this doesn't work for *sshd* under NT, even though the service appear to have stopped in the control panel. You need to kill the process manually. This can be done with the program *kill.exe* from the NT Resource Kit. Get the pid of *sshd.exe* from the NT Task Manager (suppose it is 392), and then type:

C:\> kill 392

14.4 Troubleshooting

If *ssh1* or *scp1* doesn't work as expected, use the -v (verbose) option so the client prints debugging messages as it operates. These messages may provide a clue to the problem. Now for some specific problems:

When I run ssh1, it says "You don't exist, go away!"

You probably didn't create C:\etc\passwd as directed. Also, make sure that you don't put drive specifications (C:) in the passwd file because the colon will be treated incorrectly as a field separator.

I can't create a folder called ".ssh" or any other name beginning with a period.

You can't do it with Windows's graphical interface. Use an MS-DOS shell and the *mkdir* command (*mkdir .ssh*).

ssh1 says "Could not create directory /home/.ssh".

You forgot to create C:\home.

scp1 complains that it can't find ssh1.

Is C:\usr\local\bin (or wherever you put ssh1.exe) in your MS-DOS PATH?

ssh-agent1 doesn't work. It prints Bad modes or owner for directory '/tmp/ssh-smith' and exits.
You're right. The SSH agent doesn't run under Windows, as it requires Unix domain sockets.

_I can't connect via sshd to my NT account that's in another domain._

That's right, you can't. NT _sshd_ lets you connect to local accounts only, i.e., in the domain of the local machine.\[1\]

\[1\] We've heard reports that NT _sshd_ authenticates accounts from any trusted domain, but we haven't verified this.

_I am still having problems._

Visit some of these fine sites for additional help:

http://marvin.criadvantage.com/caspian/Software/SSHD-NT/
http://www.gnac.com/techinfo/ssh_on_nt/
http://www.onlinemagic.com/~bgould/sshd.html
http://v.iki.fi/nt-ssh.html

### 14.5 Summary

Sergey Okhapkin's port of SSH1 Version 1.2.26 is, in our experience, the best Windows port of SSH1 available. It provides a working _ssh1_, _scp1_, and _ssh-keygen1_, sufficient for a typical SSH environment; and as a bonus, _sshd_ is available as well. Be aware that 1.2.26 isn't the most recent version of SSH1, however, so it may have security issues that are fixed in later official versions.
Chapter 15. SecureCRT (Windows)

SecureCRT, created by Van Dyke Technologies, is a commercial SSH client for Microsoft Windows 9x, NT, and 2000. It is structured as a terminal program; in fact, it is based on the terminal program CRT, another Van Dyke product. As a result, SecureCRT's terminal capabilities are quite configurable. It includes emulation of several terminal types, logins via Telnet as well as SSH, a scripting language, a keymap editor, SOCKS firewall support, chat features, and much more. We will focus only on its SSH capabilities, however.

SecureCRT supports both SSH-1 and SSH-2 in a single program. Other important features include port forwarding, X11 packet forwarding, and support for multiple SSH identities. It doesn't include an agent. Secure file copy is accomplished not by an scp-type program, but by ZModem, the old protocol for uploading and downloading files. (The remote machine must have ZModem installed.) If ZModem is used while you're logged in via SSH, these file transfers are secure.

We've organized this chapter to mirror the first part of the book covering Unix SSH implementations. When appropriate, we refer you to the earlier material for more detailed information.

Our discussion of SecureCRT is based on Version 3.1.2, dated September, 2000.

15.1 Obtaining and Installing

SecureCRT may be purchased and downloaded from Van Dyke Technologies:

http://www.vandyke.com/

A free evaluation version is available, expiring 30 days after installation, so you can try before you buy. Installation is straightforward and glitch-free. The software is distributed as a single .exe file; simply run it to install the program. You will need a serial number and license key to unpack the archive, and these are provided by Van Dyke to each registered user. Follow the onscreen instructions, installing the software in any folder you like. We accepted the default choices.

15.2 Basic Client Use

Once you've installed the program, it's time to set up a new session, which is SecureCRT's word for a collection of settings. Choose "Connect..." from the File menu, and in the window that appears, click the Properties button. This opens the Session Options window shown in Figure 15-1. Select Connection, and enter the information as shown in the figure. Choose password authentication for now. Click OK to close the window, and in the Connect window, click the Connect button. You should be prompted for your login password on the remote machine, and then you'll be logged in via SSH.

[1] “Session” is an unfortunate choice of term. Usually a session means an active SSH connection, not a static collection of settings.

Figure 15.1. Secure CRT Session Options window
Once you're logged in, the program operates just like a normal terminal program. SSH's end-to-end encryption is transparent to the user, as it should be.

15.3 Key Management

SecureCRT supports public-key authentication using RSA keys. It can generate keys with a built-in wizard or use existing SSH-1 and SSH-2 keys. It also distinguishes between two different types of SSH identities: global and session-specific.

15.3.1 RSA Key Generation Wizard

SecureCRT’s RSA Key Generation Wizard creates key pairs for public-key authentication. The utility is run from the Session Options window, by clicking the Advanced button, the General tab, and then Create Identity File.

Operation is straightforward. All you need to supply is the passphrase, the number of bits in the key, and some random data by moving your mouse around the screen. RSA Key Generation Wizard then creates a key pair and stores it in two files. As with the Unix SSH implementations, the private key filename is anything you choose, and its corresponding public key filename is the same with .pub added.

Once your key pair is generated, you need to copy the public key to the SSH server machine, storing it in your account’s authorization file. To accomplish this:

1. Log into the SSH server machine using SecureCRT and password authentication.
2. View the public key file and copy the full text of the key to the Windows clipboard.
3. Install the public key (by pasting from the clipboard as necessary) on the SSH server machine in your remote account. [Section 2.4.3]
4. Log out.
5. In the Session Options window, select Connection, and change Authentication from Password to RSA.
6. Log in again. SecureCRT prompts you for your public key passphrase, and you'll be logged in.

15.3.2 Using Multiple Identities

SecureCRT supports two types of SSH identities. Your global identity is the default for all SecureCRT sessions. You may override the default by using a session-specific identity that may differ (as the name implies) for each session you define.

In the Session Options window, click the Advanced button and the General tab. Under Identity Filename, you may select global and session-specific key files.

15.4 Advanced Client Use

SecureCRT lets you change settings for its SSH features and its terminal features. We will cover only the SSH-related ones. The others (and more details on the SSH features) are found in SecureCRT's online help.

SecureCRT calls a set of configuration parameters a session. It also distinguishes between session options that affect only the current session and global options that affect all sessions.

You can change session options before starting an SSH connection or while you are connected. Some options can't be changed while connected, naturally, such as the name of the remote SSH server machine. View the Session Options window (Figure 15-1) by selecting Session Options from the Options menu or clicking the Properties button on the button bar.

15.4.1 Mandatory Fields

To establish any SSH connection, you must fill in all the Connection fields in the Session Options window. These include:

Name

A memorable name for your collection of settings. This can be anything, but it defaults to the name of the SSH server.

Protocol

Either SSH-1 or SSH-2.

Hostname

The name of the remote SSH server machine to which you want to connect.

Port

The TCP port for SSH connections. Virtually all SSH clients and servers operate on port 22. Unless you plan to connect to a nonstandard SSH server, you won't need to change this. [Section 7.4.4.1]
**Username**

Your username on the remote SSH server machine. If you're using public key (RSA) authentication, this username must belong to an account that contains your public key.

**Cipher**

The encryption algorithm to be used. Unless you have strong feelings about ciphers, use the default (3DES).

**Authentication**

How you identify yourself to the SSH server. This can be password authentication (using your remote login password), RSA authentication (public key), or TIS. [Section 15.4.3] Trusted-host authentication isn't supported.

**15.4.2 Data Compression**

SecureCRT can transparently compress and uncompress the data traveling over an SSH connection. This can speed up your connection. [Section 7.4.11]

In the Session Options window, choose Connection, and click the Advanced button and the General tab. The checkbox "Use Compression" enables data compression. You may also set a value for Compression Level. Its function is identical to the CompressionLevel keyword of SSH1. The higher the value, the better the compression, but the greater load on the CPU, potentially slowing your computer.

**15.4.3 TIS Authentication**

SecureCRT can authenticate you via the Gauntlet firewall toolkit from Trusted Information Systems (TIS). [Section 5.5.1.8] In the Session Options window, under Connection, simply set Authentication to TIS.

**15.4.4 Firewall Use**

SecureCRT supports connections through several types of firewalls, such as the SOCKS4 and SOCKS5 firewalls supported by the SSH1 and SSH2 servers. Visit the Global Options window, select Firewall, and fill in the requested fields. You need to know the hostname or IP address of the firewall, and the TCP port on which to connect.

**15.5 Forwarding**

SecureCRT supports the SSH feature called forwarding (Chapter 9), in which another network connection can be passed through SSH to encrypt it. It is also called tunneling because the SSH connection provides a secure "tunnel" through which another connection may pass. Both TCP port forwarding and X forwarding are supported.

**15.5.1 Port Forwarding**

Port forwarding permits an arbitrary TCP connection to be routed through an SSH connection, transparently encrypting its data. [Section 9.2] This turns an insecure TCP connection, such as
Telnet, IMAP, or NNTP (Usenet news), into a secure one. SecureCRT supports local port forwarding, meaning that your local SSH client (SecureCRT) forwards the connection to a remote SSH server.

Each SecureCRT session you create may each have different port forwardings set up. To set up forwarding to a particular remote host, disconnect from that host (if you're connected) and open the Session Options window. Click the Advanced button and the Port Forwarding tab. Here's where you create port forwardings (see Figure 15-2).

**Figure 15.2. SecureCRT Port Forwarding tab**

To create a new forwarding, first click the New button. Then fill in the name of the remote host where the TCP service (e.g., IMAP or NNTP) is found, the remote port number for that service, and finally, a local port number (on your PC) to use for the forwarding. This can be just about any number, but for tradition's sake, make it 1024 or higher. Choose a local port number that's not being used by any other SSH client on your PC.

When you're done, click Save to save the forwarding. Then reopen your SSH connection, and your desired TCP port will be forwarded for the duration of your connection.

### 15.5.2 X Forwarding

The X Window System is the most popular windowing software for Unix machines. If you want to run remote X clients that open windows on your PC, you need:

- A remote host, running an SSH server, that has X client programs available
- An X server running on your PC under Windows, such as Hummingbird's eXceed

SSH makes your X connection secure by a process called X forwarding. Section 9.3 Turning on X forwarding is trivial in SecureCRT. Simply put a checkmark in the checkbox "Forward X11 Packets." It is found in the Session Options window: click the Advanced button, and select the Port Forwarding tab.
To secure an X connection by forwarding it through SSH, first run SecureCRT and establish a secure terminal connection to the SSH server machine. Then run your PC's X server, disabling its login features such as XDM. Now simply invoke X clients on the server machine.

15.6 Troubleshooting

SecureCRT, like any other SSH client, can run into unexpected difficulties interacting with an SSH server. In this section we cover problems specific to SecureCRT. For more general problems, see also Chapter 12.

15.6.1 Authentication

When I use an RSA key, SecureCRT says "Internal error loading private key."

SecureCRT accepts only DSA keys for SSH2 (in conformance with the current draft SSH-2 standard). Some other SSH-2 implementations also support RSA keys; these don't work with SecureCRT at present. [Section 3.9.1]

I tried loading a key generated by another SSH-2 implementation, but SecureCRT says "The private key is corrupt."

While SSH protocol draft standard specifies how keys are represented within an SSH session, it doesn't cover formats for storing those keys in files. As a result, implementors are free to make up different, incompatible formats, making your life difficult. This is true of both SSH1 and SSH2, although there is less difference among SSH1 implementations than among SSH2. Since this may change, we will just say: "contact the vendor."

After typing my passphrase, I get a dialog box, "SSH_SMSG_FAILURE: invalid SSH state," and the session disconnects without logging me in.

No pseudo-tty is being allocated. Possibly your remote account has the no-pty option set in its authorized_keys file. [Section 8.2.9]

SSH-2 authentication fails with the message "SecureCRT is disconnecting from the SSH server for the following reason: reason code 2."

At press time, some SSH-2 clients and servers from different vendors don't work together due to different interpretations of the SSH-2 draft standard. We hope such problems will have settled down by the time you read this. In the meantime, make sure you select the correct SSH Server in SecureCRT's Properties window, under Connection. To determine the type of server to which you are connecting, telnet to the server machine's SSH port (usually 22) and read the version string that appears. Using MS-DOS telnet, type:

telnet server.example.com 22

and a window opens with a response like:

SSH-1.99-2.0.13 F-SECURE SSH

indicating (in this case) F-Secure SSH Server Version 2.0.13.

15.6.2 Forwarding
I can't do port forwarding. I get a message that the port is already in use.

Do you have another SecureCRT window open and connected with the same port forwarding setup? You can't have two connections forwarding the same local port. As a workaround, create a second session that duplicates the first, but with a different local port number. Now you can use both sessions simultaneously.

15.7 Summary

We have used SecureCRT for over a year and find it a solid, stable, and capable product with strong vendor support. Its shortcomings are that it lacks an agent and scp.
Chapter 16. F-Secure SSH Client (Windows, Macintosh)

F-Secure Corporation, a distributor of commercial versions of SSH1 and SSH2 for Unix, also produces SSH clients for Windows and the Macintosh. [Section 1.5] The client product line, cleverly named F-Secure SSH Client, is a windowing, VT100-style terminal emulator that permits logins via both SSH protocols, SSH-1 and SSH-2.

For Windows, we evaluated Version 4.1, which supports both protocols in a single product. Prior versions supported SSH-1 and SSH-2 in separate products. For Macintosh, we evaluated Versions 1.0 (SSH-1) and 2.0 (SSH-2), but at press time, they didn't run reliably on our system, so we won't cover them in this book. We experienced unexplained crashes that could have been due to our Macintosh configuration or to F-Secure. We had no such difficulties with the Windows client.

F-Secure SSH Client has a fairly rich set of features. Under SSH-1, it supports remote logins, port forwarding, X forwarding, key pair generation, and a selection of terminal program features. For SSH-2 only, it includes secure file copying by *scp2* and *sftp2*.

16.1 Obtaining and Installing

F-Secure's products may be purchased online at:

http://www.f-secure.com/

Free evaluation versions are available. Installation is accomplished by a traditional Windows install program: you choose the destination directory and make a few other simple decisions.

16.2 Basic Client Use

Client behavior is set in the Properties window: visit the Edit menu and select Properties (Figure 16-1). Select Connection, and enter the information as shown in the figure. Click OK to close the window, and then in the File menu choose Connect. Since no public keys are known to the program yet, password authentication is used. If all goes well, you are prompted for your login password by the remote server, and then you're logged in via SSH.

Figure 16.1. F-Secure SSH Client Connection properties window
Once logged in, the program operates just like a normal terminal program. SSH's end-to-end encryption is transparent to the user, as it should be.

When your settings are as you like them, create a session file to store them. Session files are named with .ssh extensions and may be saved and opened with (surprise, surprise) the Save and Open commands in the File menu.

### 16.3 Key Management

F-Secure SSH Client supports public-key authentication with RSA or DSA keys. It can generate keys with its built-in Key Generation Wizard or use existing SSH-1 or SSH-2 keys.

#### 16.3.1 Generating Keys

The Key Generation Wizard is accessible within the program from the Tools menu. The wizard prompts you for the key-generation algorithm (RSA or DSA), the number of bits in the key, the key comment and passphrase, and the name of the key. After generation, the key is stored in the Windows registry and is accessible from the Properties window under User Keys. Incidentally, the Windows registry key is:

HKEY_CURRENT_USER\Software\Data Fellows\F-Secure SSH 2.0

#### 16.3.2 Importing Existing Keys

F-Secure SSH Client stores keys in the Windows registry. Most other SSH products store keys in files, so if you want to use an existing key with F-Secure, you must import it into the registry:

1. Bring up the Properties window.
2. Select User Keys.
3. Select the RSA or DSA tab, if you are importing an RSA or DSA key.
4. Select the "Import..." button.
5. Browse to your key file, and select it.
6. Type the key's passphrase (for SSH-1 format keys only).

The key is now imported into F-Secure and ready to use.

16.3.3 Installing Public Keys

For SSH-2 public keys only, F-Secure SSH Client includes the Key Registration Wizard, which automatically uploads and installs your public key on an SSH-2 server machine where your remote account resides. What a great feature! Of course the operation is secure: it connects to your remote account by SSH-2 using password authentication.

SSH-1 public keys must be installed manually on the server. Connect the remote host using password authentication, and then open the Properties window and select User Keys. From here you have two choices:

- Export your public key to a file, using the "Export..." button, then transfer the file to the remote server machine, and copy its contents into your authorized_keys file.
- Copy your public key to the Windows clipboard, using the Copy To Clipboard button, and then paste it into your remote authorized_keys file.

16.3.4 Using Keys

Unlike most Unix SSH products, F-Secure SSH Client doesn't let you specify which key to use for a session. Instead, it tries each key in turn. When one matches a public key on the server, you are prompted for your passphrase. To reject F-Secure's choice of key and use a different one, press the Escape key or click Cancel, and the next key is selected and tried. If all keys fail, the program falls back to password authentication.

16.4 Advanced Client Use

To establish an SSH connection, you must fill in the following Connection fields in the Properties window. These include:

**Host Name**

The name of the remote SSH server machine to which you want to connect.

**User Name**

Your username on the remote SSH server machine. If you're using public key (RSA) authentication, this username must belong to an account that contains your public key.

**Port Number**

The TCP port for SSH connections. Virtually all SSH clients and servers operate on port 22. Unless you plan to connect to a nonstandard SSH server, you don't need to change this. [Section 7.4.4.1]

**SSH Protocol**
You may require SSH-1 or SSH-2, or select Automatic to let the program figure it out based on the server’s response.

Optionally, you may select the encryption cipher and an authentication method. In the Properties window, select Cipher to choose the set of encryption ciphers you will permit your client to use. (The default should be acceptable for most uses.) The SSH server negotiates with your client to choose a cipher they both support.

Your authentication method can be public key or password, which may be chosen in the login window. The program automatically tries to authenticate with each of your User Keys in order. [Section 16.3.4]

### 16.4.1 Data Compression

F-Secure SSH can transparently compress and uncompress the data traveling over an SSH connection, which can speed up your connection. [Section 7.4.11]

In the Properties window, choose Connection, and check the box labeled Compression. There's no way to set different compression levels as in SSH1.

### 16.4.2 Debugging with Verbose Mode

Is your SSH session not working as expected? Turn on verbose mode so status messages are printed in the window as your session runs. This can help you locate and solve problems.

In the Properties window, select Appearance and check the box labeled Verbose Mode. The next time you connect, you'll see messages like this:

```
debug: connecting ...
debug: addresses 219.243.169.50
debug: Registered connecting socket: 12
debug: Connection still in progress
debug: Marked name resolver 1 killed
debug: Replaced connected socket object 12 with a stream
```

Verbose mode is much like its counterpart in the Unix SSH products. [Section 7.4.15] It can be an indispensable tool for diagnosing problems with your connection.

### 16.4.3 SOCKS Proxy Server

F-Secure SSH Client supports connecting through SOCKS Version 4 proxy servers. [Section 4.1.5.8] On the Properties window, select Socks and fill in the hostname or IP address of the proxy server machine and the port number on the proxy (the usual SOCKS port is 1080).

### 16.4.4 Accepting Host Keys

Every SSH server has a unique host key that represents the server’s identity, so SSH clients can verify that they are speaking with the actual server and not an impostor. [Section 2.3.1] F-Secure SSH Client keeps track of all host keys it encounters. The keys are stored in the Windows registry.

If you want F-Secure SSH Client to reject host keys it hasn’t seen before, visit the Properties window and select Security. A checkbox is available to set this option.

### 16.4.5 Additional Security Features
Normally, F-Secure SSH Client keeps track of hostnames, usernames, filenames, and terminal input and output that it encounters. Any time you want to purge this information from the program (say, to prevent a third party from viewing it on your computer), visit the Properties window and select Security. The buttons on this window will delete the information.

16.4.6 Secure File Transfer with SFTP

A graphical file transfer program, F-Secure SSH FTP, is also included. Its user interface should be familiar to anyone who has used a graphical FTP client, except that you must set up authentication via SSH. We don't document this program, as it comes with online help, but we did want to mention it.

16.4.7 Command-Line Tools

F-Secure SSH comes with a graphical terminal program as well as command-line clients using the SSH-2 protocol. These include ssh2, scp2, and sftp2. These programs are much the same as their counterparts in SSH2, as described in Chapter 2, except:

- Some of the Unix command-line options aren't supported. Type the program name by itself (e.g., ssh2) to see the current list of options.
- Key files aren't supported; these programs read keys from the Windows registry, just as F-Secure SSH Client does.

The command-line programs are useful for scripting and batch files, or for executing remote commands on the server machine:

```
C:/> ssh2 server.example.com mycommand
```

16.5 Forwarding

F-Secure SSH Client supports forwarding (Chapter 9), in which another network connection can be passed through SSH to encrypt it. It is also called tunneling because the SSH connection provides a secure "tunnel" through which another connection may pass. Both TCP port forwarding (local and remote) and X forwarding are supported.

16.5.1 Port Forwarding

Each F-Secure configuration you create may have different ports forwarded. To set up forwarding to a particular remote host, disconnect from that host (if you're connected), open the Properties window, and notice the Tunneling category. Select Local Tunneling to set up local forwarding (see Figure 16-2), or Remote Tunneling for remote forwarding, respectively. [Section 9.2.3] Either way, you are prompted for similar information:

*Source Port:*

The local port number

*Destination Host:*

The remote hostname
**Destination Port:**

The remote port number

**Application to Start:**

An external application to be launched when this port is forwarded

**Figure 16.2. F-Secure SSH Client local port forwarding options**

For example, to tunnel a *telnet* connection (TCP port 23) through SSH to connect to `server.example.com`, you can specify:

- **Source port:** 8500 (any random port number)
- **Destination Host:** `server.example.com`
- **Destination Port:** 23
- **Application to Start:** `c:\windows\telnet.exe`

Once you've made your choices, reopen the SSH connection, and the ports will be forwarded for the duration of your connection.

Note that F-Secure SSH Client forbids remote connections to locally forwarded ports. This security feature is analogous to specifying "GatewayPorts no". [Section 9.2.1.1]

### 16.5.2 X Forwarding

The X Window System is the most popular windowing software for Unix machines. If you want to run remote X clients that open windows on your PC, you need:

- A remote host, running an SSH server, that has X client programs available
- An X server running on your PC under Windows, such as Hummingbird's eXceed

SSH makes your X connection secure by a process called X forwarding. [Section 9.3] Turning on X forwarding is trivial in F-Secure SSH Client: open the Properties window, select Tunneling, and put a checkmark in the box Enable X11 Tunneling. You may also select the X display number, which you also may change during your SSH session.
To secure an X connection by forwarding it through SSH, first run F-Secure SSH Client and establish a secure terminal connection to the SSH server machine. Then run your PC’s X server, disabling its login features such as XDM. Now simply invoke X clients on the server machine, and their windows will open on your local X display.

16.6 Troubleshooting

F-Secure SSH Client, like any other SSH client, can run into unexpected difficulties interacting with an SSH server. In this section we cover problems specific to F-Secure SSH. For more general problems, see also Chapter 12.

Why is scrolling so slow for F-Secure SSH for Windows?

At press time, F-Secure's scrolling speed is quite poor, but you can speed it up somewhat by turning on jump scrolling, which is disabled by default. It can be enabled only through F-Secure's key-mapping facility. The following setup enables jump scrolling for one given Session File:

1. In your F-Secure installation directory, locate the file `Keymap.map`.
2. Edit this file and locate the line containing "enable fast-scroll-mapping".
3. Uncomment this line by removing any "#" symbols at the beginning of the line.
4. Save and close the file.
5. In the Edit menu, select Properties.
6. In the Properties window, select Keyboard.
7. Under Map Files, in the Keyboard blank, enter the path to the `Keymap.map` file you edited.
8. Click OK.

You can now toggle between smooth and jump scrolling by pressing Control-Alt-F3. Press this key sequence once in your F-Secure SSH session, and your scrolling will become speedy.

I got the error message "Warning: Remote host failed or refused to allocate a pseudo tty."

Your server's SSH-1 authorized_keys file might have the no-pty option specified for the corresponding public key. [Section 8.2.9]

I can't do port forwarding. I get a message that the port is already in use.

Do you have another F-Secure window open and connected with the same port forwarding setup? You can't have two connections forwarding the same local port. As a workaround, create a second configuration that duplicates the first but with a different local port number. Now you can use both connections simultaneously.

I tried using the Key Registration Wizard but it didn't work. The Wizard said "Disconnected, connection lost."

First, verify that your hostname, remote username, and remote password are given correctly. If they are definitely correct, click the Advanced button and check the information there. Is the SSH-2 server running on the default port, 22, or a different one? Do the SSH2 directory and authorization filenames you provided match the ones on the server?
Another possible (and more technical) cause is a missing or incomplete `sftp-server` subsystem entry in the server's `/etc/sshd2_config` file. F-Secure recommends that the fully qualified path to `sftp-server2` appear in this entry, or else the SSH server possibly can't locate it. The Key Registration Wizard uses the SFTP protocol to transfer key files. The system administrator of the server machine needs to make this change.

**How do I prevent the initial welcome window from being displayed in F-Secure SSH2?**

For Windows, modify the Registry value:

```
HKEY_CURRENT_USER\Software\Data Fellows\F-Secure SSH 2.0\TnT\Settings\ShowSplash
```

The default is `yes`; set it to `no`. The next time you run the program, no welcome window appears. (This is an undocumented feature.)

**How do I eliminate the Logon Information window that pops up when I run F-Secure SSH2 Client? I'm using public-key authentication so the window seems unnecessary.**

In the Properties window, select Connection, and check the box Auto- Connect On Open.

### 16.7 Summary

Overall, F-Secure SSH Client for Windows is a good, solid product. The present version does have some long-standing weaknesses: no agent, no secure file copy in the SSH-1 products, incomplete documentation (almost no mention of keyboard map files), and (though `sftp` is supported in the SSH-2 products) the inability to turn on jump scrolling in the Preferences window. Regardless, we've used the product reliably in various versions for over a year. F-Secure SSH is still in active development, so some of its limitations may still be overcome.
Chapter 17. NiftyTelnet SSH (Macintosh)

NiftyTelnet SSH is a minimal, freely distributable SSH client for the Macintosh by Jonas Walldén. Based on NiftyTelnet by Chris Newman, NiftyTelnet SSH is a graphical terminal program with SSH-1 support added. It supports remote logins and secure file copying. It also remembers your public key passphrase (i.e., caching it in memory) if you open multiple terminal windows. This isn't an agent, however.

NiftyTelnet's best features are that it is free, and it works pretty well. On the negative side, it doesn't support forwarding of any kind, and it can't generate SSH key pairs. In order to use public-key authentication, you need another SSH program to generate keys for your identity, such as ssh-keygen1 from SSH1.

Our discussion of NiftyTelnet SSH is based on Version 1.1 R3.

17.1 Obtaining and Installing

NiftyTelnet SSH can be downloaded from:

http://www.lysator.liu.se/~jonasw/freeware/niftyssh/

and unpacked with Stuffit Expander into a folder of your choice. You may copy the folder to any location on your Macintosh.

When NiftyTelnet SSH is run for the first time, it presents the New Connection dialog box shown in Figure 17-1. Click the New button to enter the settings for an SSH client/server connection. Figure 17-2 highlights the important fields for configuring SSH: Host Name, Protocol, and RSA Key File. For the Host Name, enter the name of a remote host running an SSH server, and set the Protocol to an SSH encryption algorithm (DES, 3DES, or Blowfish). If you plan to use password authentication, leave the RSA Key File line empty. Otherwise, if you have already installed a private key file on your Mac, fill in the location. You must list the entire path to the file, with folder names separated by colons. For example, if your key file Identity is found by opening the disk MyDisk, then the folder SSH, and then the folder NiftyTelnet, enter:

MyDisk:SSH:NiftyTelnet:Identity

Once your settings are complete, connect to the remote host using password authentication. Copy your public key onto the remote host, log out, and reconnect using public-key authentication.

Figure 17.1. NiftyTelnet New Connection dialog box
17.2 Basic Client Use

NiftyTelnet SSH began life as NiftyTelnet, a Macintosh Telnet application, with SSH support added later by another programmer. Most of the configurable parameters relate to Telnet, so we don't cover them, just the SSH-specific ones.

![Figure 17.2. NiftyTelnet SSH Settings window](image)

17.2.1 Authentication

For SSH, you specify only your cipher (labeled "Protocol") and the path to your private key file (labeled "RSA Key File"), as shown in Figure 17.2. The default authentication method is public key, but if this fails or you have no key file, it falls back to password authentication.

The only tricky part is the path, which must be typed, rather than browsed by the usual Macintosh file selector. [Section 17.3]

17.2.2 Scp
The Scp button on the New Connection dialog box permits secure copying of files and folders via SSH between your Mac and a remote computer. This feature operates much like the scp1 client of SSH1 but with a graphical interface (see Figure 17-3).

Figure 17.3. NiftyTelnet Scp window

Local files and folders may be selected by browsing, but remote files and folders must have their names entered manually. If you're used to Mac FTP clients such as Fetch, this interface might seem a bit spartan. Nonetheless, it works, and if you ask NiftyTelnet SSH to remember your password, you don't have to retype it for each file transfer.

17.2.3 Host Keys

Every SSH server has a unique host key that represents the server's identity, so clients such as NiftyTelnet SSH can verify that they are speaking with the actual server and not an impostor. [Section 2.3.1]

NiftyTelnet keeps track of all server host keys it encounters. The keys are stored in a file called NiftyTelnet SSH Known Hosts in your Macintosh's Preferences folder within the System folder. The file has the same format as SSH1 known hosts files. [Section 3.5.2.1]

17.3 Troubleshooting

I want to do public-key authentication. How do I generate a key pair with NiftyTelnet SSH?

You can't. You must generate the key pair with a different SSH program, such as ssh-keygen1 from SSH1 on a Unix machine. [Section 2.4.2]

What do I put in the "RSA Key File" blank?

The complete path to your private key file, folder by folder, with the folder names separated by colons. For example, if your key file Identity is found by opening the disk MyDisk, then the folder SSH, and then the folder NiftyTelnet, enter:

MyDisk:SSH:NiftyTelnet:Identity
I cannot get the Scp feature to work. When I try to transfer a file, a window labeled “File: Waiting For Connection” appears for a few seconds and then disappears. No file is transferred.

According to the author, this problem sometimes occurs if NiftyTelnet SSH is communicating with an old SSH server. Reportedly, sshd 1.2.25 and newer should work. We saw this problem initially with a 1.2.27 server, but it went away on its own, and we couldn't figure out the cause.

### 17.4 Summary

NiftyTelnet SSH is a fine choice for a Macintosh SSH client. It supports scp, a rarity among SSH-1 implementations for PCs and Macs, and it conveniently remembers your passphrase until the program exits. On the other hand, NiftyTelnet SSH has the fewest features of any SSH implementation we’ve seen and can’t generate key pairs. If you need more bells and whistles, such as port forwarding and a wider set of options, consider F-Secure SSH Client for the Mac (Chapter 16).
Appendix A. SSH2 Manpage for sshregex

SSHREGEX(1) SSH2

Description

This document describes the regular expressions (or globbing patterns) used in filename globbing with scp2 and sftp2.

Patterns

The escape character is a backslash \"\". With this you can escape meta characters which you'd like to use in their plain character form.

In the following examples literal "E" and "F" denote any expression, be it a pattern or character, etc.

*  
Match any string consisting of zero or more characters. The characters can be any characters apart from slashes (/). However, the asterisk does not match a string if the string contains a dot (.) as its first character, or if the string contains a dot immediately after a slash. This means that the asterisk cannot be used to match filenames that have a dot as their first character.

If the previous character is a slash (/), or the asterisk (*) is used to denote a match at the beginning of a string, it does match a dot (.).

That is, the "*" functions as is normal in UNIX shell fileglobs.

?  
Match any single character except for a slash (/). However, do not match a dot (.) if located at the beginning of the string, or if the previous character is a slash (/).

That is, "?" functions as is normal in Unix shell fileglobs (at least ZSH, although discarding the dot may not be a standard procedure).

/**  
Match any sequence of characters that is either empty, or ends in a slash. However, the substring "/." is not allowed. This mimics ZSH's ingenious /** construct. (Observe that "**" is equivalent to "*".)

E#  
Act as Kleene star, match E zero or more times.
Closure, match E one or more times.

( Start a capturing subexpression.

) End a capturing subexpression.

Disjunction, match (inclusively) either E or F. E is preferred if both match.

Start a character set. (See below).

**Character Sets**

A character set starts with "[" and ends at nonescaped "]" that is not part of a POSIX character set specifier and that does not follow immediately after "]".

The following characters have a special meaning and need to be escaped if meant literally:

- (minus sign) A range operator, except immediately after "]", where it loses its special meaning.

- ^ or ! If immediately after the starting "]", denotes a complement: the whole character set will be complemented. Otherwise literal. "^".

[:alnum:] Characters for which "isalnum" returns true (see ctype.h).

[:alpha:] Characters for which "isalpha" returns true (see ctype.h).

[:cntrl:] Characters for which "iscntrl" returns true (see ctype.h).

[:digit:] Characters for which "isdigit" returns true (see ctype.h).

[:graph:]
Characters for which "isgraph" returns true (see ctype.h).

[:lower:]
Characters for which "islower" returns true (see ctype.h).

[:print:]
Characters for which "isprint" returns true (see ctype.h).

[:punct:]
Characters for which "ispunct" returns true (see ctype.h).

[:space:]
Characters for which "isspace" returns true (see ctype.h).

[:upper:]
Characters for which "isupper" returns true (see ctype.h).

[:xdigit:]
Characters for which "isxdigit" returns true (see ctype.h).

Example

[[:xdigit:]XY] is typically equivalent to [0123456789ABCDEFabcdefXY].

Authors

SSH Communications Security Corp.

For more information, see http://www.ssh.com/.

See also

scp2 (1), sftp2 (1)
Appendix B. SSH Quick Reference

Section 2.1. Legend

Section 2.2. sshd Options

Section 2.3. sshd Keywords

Section 2.4. ssh and scp Keywords

Section 2.5. ssh Options

Section 2.6. scp Options

Section 2.7. ssh-keygen Options

Section 2.8. ssh-agent Options

Section 2.9. ssh-add Options

Section 2.10. Identity and Authorization Files

Section 2.11. Environment Variables

2.1 Legend

<table>
<thead>
<tr>
<th>Mark</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>Yes: feature is supported/included</td>
</tr>
<tr>
<td>1</td>
<td>SSH-1 protocol only, not SSH-2</td>
</tr>
<tr>
<td>2</td>
<td>SSH-2 protocol only, not SSH-1</td>
</tr>
<tr>
<td>F</td>
<td>F-Secure SSH only</td>
</tr>
<tr>
<td>N</td>
<td>Not in F-Secure SSH</td>
</tr>
</tbody>
</table>

2.2 sshd Options

<table>
<thead>
<tr>
<th>SSH1</th>
<th>SSH2</th>
<th>OpenSSH</th>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td></td>
<td>✓</td>
<td>-4</td>
<td>Use IPv4 addresses only</td>
</tr>
<tr>
<td></td>
<td>✓</td>
<td>✓</td>
<td>-6</td>
<td>Use IPv6 addresses only</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-b bits</td>
<td># of bits in server key</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-d</td>
<td>Verbose mode</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td></td>
<td>-d level</td>
<td>Enable debug messages</td>
</tr>
<tr>
<td></td>
<td>✓</td>
<td>✓</td>
<td>-d &quot;module=level&quot;</td>
<td>Enable debug messages per module</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-f filename</td>
<td>Use other configuration file</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-g time</td>
<td>Set login grace time</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-h filename</td>
<td>Use other host key file</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-i</td>
<td>Use inetd for invocation</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>--------------------------</td>
</tr>
<tr>
<td>✓</td>
<td></td>
<td></td>
<td>-k time</td>
<td>Key regeneration interval</td>
</tr>
<tr>
<td>✓</td>
<td></td>
<td></td>
<td>-o &quot;keyword value&quot;</td>
<td>Set configuration keyword</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-p port</td>
<td>Select TCP port number</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-q</td>
<td>Quiet mode</td>
</tr>
<tr>
<td>✓</td>
<td></td>
<td>✓</td>
<td>-Q</td>
<td>Quiet if RSA support is missing</td>
</tr>
<tr>
<td>✓</td>
<td></td>
<td>✓</td>
<td>-v</td>
<td>Verbose mode</td>
</tr>
<tr>
<td>✓</td>
<td></td>
<td>✓</td>
<td>-V</td>
<td>Print version number</td>
</tr>
<tr>
<td>✓</td>
<td></td>
<td>✓</td>
<td>-V id</td>
<td>OpenSSH SSH2 compatibility mode</td>
</tr>
</tbody>
</table>

### 2.3 sshd Keywords

<table>
<thead>
<tr>
<th>SSH1</th>
<th>SSH2</th>
<th>OpenSSH</th>
<th>Keyword</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>#</td>
<td>Any text</td>
<td>Comment line</td>
</tr>
<tr>
<td>✓</td>
<td></td>
<td></td>
<td>AccountExpireWarningDays</td>
<td># days</td>
<td>Warn user of expiration</td>
</tr>
<tr>
<td>✓</td>
<td></td>
<td>✓</td>
<td>AFSTokenPassing</td>
<td>Yes/no</td>
<td>Forward AFS tokens to server</td>
</tr>
<tr>
<td>N</td>
<td></td>
<td></td>
<td>AllowAgentForwarding</td>
<td>Yes/no</td>
<td>Enable agent forwarding</td>
</tr>
<tr>
<td>✓</td>
<td></td>
<td></td>
<td>AllowedAuthentications</td>
<td>Auth types</td>
<td>Permitted authentication techniques</td>
</tr>
<tr>
<td>N</td>
<td></td>
<td>✓</td>
<td>AllowCshrcSourcingWithSubsystems</td>
<td>Yes/no</td>
<td>Source shell startup file</td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td>AllowForwardingPort</td>
<td>Port list</td>
<td>Permit forwarding for ports</td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td>AllowForwardingTo</td>
<td>Host/port list</td>
<td>Permit forwarding for hosts</td>
</tr>
<tr>
<td>✓</td>
<td>N</td>
<td>✓</td>
<td>AllowGroups</td>
<td>Group list</td>
<td>Access control by Unix group</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td></td>
<td>AllowHosts</td>
<td>Host list</td>
<td>Access control by hostname</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td></td>
<td>AllowSHosts</td>
<td>Host list</td>
<td>Access control via .shosts</td>
</tr>
<tr>
<td>✓</td>
<td>N</td>
<td>✓</td>
<td>AllowTcpForwarding</td>
<td>Yes/no</td>
<td>Enable TCP port forwarding</td>
</tr>
<tr>
<td>N</td>
<td></td>
<td></td>
<td>AllowTcpForwardingFor-Users</td>
<td>User list</td>
<td>Per user forwarding</td>
</tr>
<tr>
<td>N</td>
<td></td>
<td></td>
<td>AllowTcpForwardingForGroups</td>
<td>Group list</td>
<td>Per group forwarding</td>
</tr>
<tr>
<td>✓</td>
<td>N</td>
<td>✓</td>
<td>AllowUsers</td>
<td>User list</td>
<td>Access control by username</td>
</tr>
<tr>
<td>N</td>
<td></td>
<td></td>
<td>AllowX11Forwarding</td>
<td>Yes/no</td>
<td>Enable X forwarding</td>
</tr>
<tr>
<td>✓</td>
<td></td>
<td></td>
<td>AuthorizationFile</td>
<td>Filename</td>
<td>Location of</td>
</tr>
<tr>
<td>Feature</td>
<td>Value</td>
<td>Description</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>-------</td>
<td>--------------------------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>authorization file</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>CheckMail</strong></td>
<td>Yes</td>
<td>Check new mail on login</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ChRootGroups</strong></td>
<td>No</td>
<td>Run <code>chroot()</code> on login</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ChRootUsers</strong></td>
<td>No</td>
<td>Run <code>chroot()</code> on login</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Ciphers</strong></td>
<td>2</td>
<td>Select encryption ciphers</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>DenyForwardingPort</strong></td>
<td>F</td>
<td>Forbid forwarding for ports</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>DenyForwardingTo</strong></td>
<td>F</td>
<td>Forbid forwarding for hosts</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>DenyGroups</strong></td>
<td>N</td>
<td>Access control by Unix group</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>DenyHosts</strong></td>
<td></td>
<td>Access control by hostname</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>DenySHosts</strong></td>
<td></td>
<td>Access control via <code>.shosts</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>DenyTCPForwardingFor-Users</strong></td>
<td>N</td>
<td>Per user forwarding</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>DenyTCPForwardingForGroups</strong></td>
<td>N</td>
<td>Per group forwarding</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>DenyUsers</strong></td>
<td>N</td>
<td>Access control by username</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>DSAAuthentication</strong></td>
<td>2</td>
<td>Permit SSH-2 DSA authentication</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>FascistLogging</strong></td>
<td></td>
<td>Verbose mode</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ForcedEmptyPasswdChange</strong></td>
<td></td>
<td>Change password if empty</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ForcedPasswdChange</strong></td>
<td></td>
<td>Change password on first login</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ForwardAgent</strong></td>
<td></td>
<td>Enable agent forwarding</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ForwardX11</strong></td>
<td></td>
<td>Enable X forwarding</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>GatewayPorts</strong></td>
<td></td>
<td>Gateway all locally forwarded ports</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>HostDSAKey</strong></td>
<td>2</td>
<td>Location of DSA key file</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>HostKey</strong></td>
<td></td>
<td>Location of host key file</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Hostkeyfile</strong></td>
<td></td>
<td>Location of host key file</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>IdleTimeout</strong></td>
<td></td>
<td>Set idle timeout</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>IgnoreRhosts</strong></td>
<td></td>
<td>Ignore <code>.rhosts</code> files</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>IgnoreRootRhosts</strong></td>
<td></td>
<td>Ignore <code>./rhosts</code> file</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>IgnoreUserKnownHosts</strong></td>
<td></td>
<td>Ignore user's known-hosts keys</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Option</td>
<td>Setting</td>
<td>Description</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------------------------------</td>
<td>------------------</td>
<td>-------------------------------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KeepAlive</td>
<td>Yes/no</td>
<td>Send keepalive packets</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KerberosAuthentication</td>
<td>Yes/no</td>
<td>Permit Kerberos authentication</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KerberosOrLocalPasswd</td>
<td>Yes/no</td>
<td>Kerberos fallback authentication</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KerberosTgtPassing</td>
<td>Yes/no</td>
<td>Support ticket-granting-tickets</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KerberosTicketCleanup</td>
<td>Yes/no</td>
<td>Destroy ticket cache on logout</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KeyRegenerationInterval</td>
<td>Time</td>
<td>Key regeneration interval</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ListenAddress</td>
<td>IP address</td>
<td>Listen on given interface</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LoginGraceTime</td>
<td>Time</td>
<td>Time limit for authentication</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LogLevel</td>
<td>Syslog level</td>
<td>Set syslog level</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Macs</td>
<td>Algorithm</td>
<td>Select MAC algorithm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MaxBroadcastsPerSecond</td>
<td># broadcasts</td>
<td>Listen for UDP broadcasts</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MaxConnections</td>
<td># connections</td>
<td>Maximum # of simultaneous connections</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NoDelay</td>
<td>Yes/no</td>
<td>Enable Nagle algorithm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PasswordAuthentication</td>
<td>Yes/no</td>
<td>Permit password authentication</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PasswordGuesses</td>
<td># guesses</td>
<td>Limit # of password tries</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PasswordExpireWarningDays</td>
<td># days</td>
<td>Warn user before expiration</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PermitEmptyPasswords</td>
<td>Yes/no</td>
<td>Permit empty passwords</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PermitRootLogin</td>
<td>Yes/no/nopwd</td>
<td>Permit superuser logins</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PGPPublicKeyFile</td>
<td>Filename</td>
<td>Default location of PGP public key file for authentication</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PidFile</td>
<td>Filename</td>
<td>Location of pid file</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Port</td>
<td>Port number</td>
<td>Select server port number</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PrintMotd</td>
<td>Yes/no</td>
<td>Print message of the day</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Protocol</td>
<td>1/2/1,2</td>
<td>Permit SSH-1 SSH-2 connections</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PublicKeyAuthentication</td>
<td>Yes/no</td>
<td>Permit public-key authentication</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PublicHostKeyFile</td>
<td>Filename</td>
<td>Location of public</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Feature</td>
<td>Value</td>
<td>Description</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------------------------------</td>
<td>-------------</td>
<td>--------------------------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QuietMode</td>
<td>Yes/no</td>
<td>Quiet mode</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RandomSeed</td>
<td>Filename</td>
<td>Location of random seed file</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RandomSeedFile</td>
<td>Filename</td>
<td>Location of random seed file</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RekeyIntervalSeconds</td>
<td>Seconds</td>
<td>Frequency of rekeying</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RequireReverseMapping</td>
<td>Yes/no</td>
<td>Do reverse DNS lookup</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RequiredAuthentications</td>
<td>Auth types</td>
<td>Required authentication techniques</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RhostsAuthentication</td>
<td>Yes/no</td>
<td>Permit .rhosts authentication</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RhostsPubKey-Authentication</td>
<td>Yes/no</td>
<td>Permit combined authentication</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RhostsRSAAuthentication</td>
<td>Yes/no</td>
<td>Permit combined authentication</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RSAAuthentication</td>
<td>Yes/no</td>
<td>Permit public-key authentication</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ServerKeyBits</td>
<td># bits</td>
<td># of bits in server key</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SkeyAuthentication</td>
<td>Yes/no</td>
<td>Permit S/Key authentication</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ssh1Compatibility</td>
<td>Yes/no</td>
<td>Enable SSH1 compatibility</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sshd1Path</td>
<td>Filename</td>
<td>Path to sshd1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SilentDeny</td>
<td>Yes/no</td>
<td>DenyHosts prints no message</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>StrictModes</td>
<td>Yes/no</td>
<td>Strict file/directory permissions</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SyslogFacility</td>
<td>Syslog level</td>
<td>Set syslog level</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TISAAuthentication</td>
<td>Yes/no</td>
<td>Permit TIS authentication</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Umask</td>
<td>Unix umask</td>
<td>Set login umask</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UseLogin</td>
<td>Yes/no</td>
<td>Select login program</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UserConfigDirectory</td>
<td>Directory name</td>
<td>Location of user SSH2 directories</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UserKnownHosts</td>
<td>Yes/no</td>
<td>Respect ~/.ssh2/knownhosts</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VerboseMode</td>
<td>Yes/no</td>
<td>Verbose mode</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X11Forwarding</td>
<td>Yes/no</td>
<td>Enable X forwarding</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X11DisplayOffset</td>
<td># offset</td>
<td>Limit X displays for SSH</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XAuthLocation</td>
<td>Filename</td>
<td>Location of xauth</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

418
## 2.4 ssh and scp Keywords

<table>
<thead>
<tr>
<th>SSH1</th>
<th>SSH2</th>
<th>OpenSSH</th>
<th>Keyword</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>#</td>
<td>Any text</td>
<td>Comment line</td>
</tr>
<tr>
<td>✓</td>
<td></td>
<td></td>
<td>AFSTokenPassing</td>
<td>Yes/no</td>
<td>Forward AFS tokens to server</td>
</tr>
<tr>
<td>N</td>
<td></td>
<td></td>
<td>AllowAgentForwarding</td>
<td>Yes/no</td>
<td>Enable agent forwarding</td>
</tr>
<tr>
<td>✓</td>
<td></td>
<td></td>
<td>AllowedAuthentications</td>
<td>Auth types</td>
<td>Permitted authentication techniques</td>
</tr>
<tr>
<td>N</td>
<td></td>
<td></td>
<td>AuthenticationNotify</td>
<td>Yes/no</td>
<td>Print message on stdout on successful authentication</td>
</tr>
<tr>
<td>N</td>
<td></td>
<td></td>
<td>AuthenticationSuccessMsg</td>
<td>Yes/no</td>
<td>Print message on stderr on successful authentication</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>BatchMode</td>
<td>Yes/no</td>
<td>Disable prompting</td>
</tr>
<tr>
<td>✓</td>
<td></td>
<td></td>
<td>CheckHostIP</td>
<td>Yes/no</td>
<td>Detect DNS spoofing</td>
</tr>
<tr>
<td>✓</td>
<td></td>
<td></td>
<td>Cipher</td>
<td>Cipher</td>
<td>Request encryption cipher</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td></td>
<td>Ciphers</td>
<td>Cipher_list</td>
<td>Supported encryption ciphers</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>ClearAllForwardings</td>
<td>Yes/no</td>
<td>Ignore any specified forwarding</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Compression</td>
<td>Yes/no</td>
<td>Enable data compression</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td></td>
<td>CompressionLevel</td>
<td>0-9</td>
<td>Select compression algorithm</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>ConnectionAttempts</td>
<td># attempts</td>
<td># of retries by client</td>
</tr>
<tr>
<td>N</td>
<td></td>
<td></td>
<td>DefaultDomain</td>
<td>Domain</td>
<td>Specify domain name</td>
</tr>
<tr>
<td>✓</td>
<td></td>
<td></td>
<td>DontReadStdin</td>
<td>Yes/no</td>
<td>Redirect stdin from /dev/null</td>
</tr>
<tr>
<td>✓</td>
<td></td>
<td></td>
<td>DSAAuthentication</td>
<td>Yes/no</td>
<td>Permit SSH-2 DSA authentication</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>EscapeChar</td>
<td>Character</td>
<td>Set escape character (^ = Ctrl key)</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td></td>
<td>FallBackToRsh</td>
<td>Yes/no</td>
<td>Use rsh if ssh fails</td>
</tr>
<tr>
<td>✓</td>
<td></td>
<td></td>
<td>ForcePTYAllocation</td>
<td>Yes/no</td>
<td>Allocate a pseudo-tty</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>ForwardAgent</td>
<td>Yes/no</td>
<td>Enable agent forwarding</td>
</tr>
<tr>
<td>✓</td>
<td></td>
<td></td>
<td>ForwardX11</td>
<td>Yes/no</td>
<td>Enable X forwarding</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>GatewayPorts</td>
<td>Yes/no</td>
<td>Gateway locally forwarded ports</td>
</tr>
<tr>
<td>✓</td>
<td></td>
<td></td>
<td>GlobalKnownHostsFile</td>
<td>Filename</td>
<td>Location of global known hosts file</td>
</tr>
<tr>
<td>✓</td>
<td></td>
<td></td>
<td>GlobalKnownHostsFile2</td>
<td>Filename</td>
<td>Location of global known hosts file</td>
</tr>
<tr>
<td>✓</td>
<td></td>
<td></td>
<td>GoBackground</td>
<td>Yes/no</td>
<td>Fork into background</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Host</td>
<td>Pattern</td>
<td>Begin section for this host</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>HostName</td>
<td>Hostname</td>
<td>Real name of host</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>✓</td>
<td></td>
<td>1</td>
<td>IdentityFile</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Filename</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Name of private key file (RSA)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>IdentityFile2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Filename</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Name of private key file (DSA)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>✓ ✓ ✓</td>
<td></td>
<td></td>
<td>KeepAlive</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Yes/no</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Send keepalive packets</td>
<td></td>
<td></td>
</tr>
<tr>
<td>✓ ✓ ✓</td>
<td></td>
<td></td>
<td>KerberosAuthentication</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Yes/no</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Permit Kerberos authentication</td>
<td></td>
<td></td>
</tr>
<tr>
<td>✓ ✓ ✓</td>
<td></td>
<td></td>
<td>KerberosTgtPassing</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Yes/no</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Support ticket-granting-tickets</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>N</td>
<td>LocalForward</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Port, socket</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Local port forwarding</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>N</td>
<td>Macs</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Algorithm</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Select MAC algorithm</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>NoDelay</td>
<td>Yes/no</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Enable Nagle algorithm</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>NumberOfPasswordPrompts</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td># prompts</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td># of prompts before failure</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>PasswordAuthentication</td>
<td>Yes/no</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Permit password authentication</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>PasswordPrompt</td>
<td>String</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Password prompt</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>PasswordPromptHost</td>
<td>Yes/no</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Hostname for password prompt</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>PasswordPromptLogin</td>
<td>Yes/no</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Username for password prompt</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Port</td>
<td>Port number</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Select server port number</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ProxyCommand</td>
<td>Command</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Connect to proxy server</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>QuietMode</td>
<td>Yes/no</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Quiet mode</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>RandomSeedFile</td>
<td>Filename</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Location of random seed file</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>RemoteForward</td>
<td>Port, socket</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Remote port forwarding</td>
<td></td>
<td></td>
</tr>
<tr>
<td>✓ ✓ ✓</td>
<td></td>
<td></td>
<td>RhostsAuthentication</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Yes/no</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Permit .rhosts authentication</td>
<td></td>
<td></td>
</tr>
<tr>
<td>✓</td>
<td></td>
<td></td>
<td>RhostsPubKey-Authentication</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Yes/no</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Permit combined authentication</td>
<td></td>
<td></td>
</tr>
<tr>
<td>✓ ✓ ✓</td>
<td></td>
<td></td>
<td>RhostsRSAAuthentication</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Yes/no</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Permit combined authentication</td>
<td></td>
<td></td>
</tr>
<tr>
<td>✓ ✓ ✓</td>
<td></td>
<td></td>
<td>RSAAuthentication</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Yes/no</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Permit public-key authentication</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N</td>
<td></td>
<td>PGPSecretKeyfile</td>
<td>Filename</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Default location of PGP private key file for authentication</td>
<td></td>
<td></td>
</tr>
<tr>
<td>✓</td>
<td></td>
<td></td>
<td>SkeyAuthentication</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Yes/no</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Permit S/Key authentication</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N</td>
<td></td>
<td>SocksServer</td>
<td>Server</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Specify SOCKS server</td>
<td></td>
<td></td>
</tr>
<tr>
<td>✓</td>
<td></td>
<td>Ssh1AgentCompatibility</td>
<td>Yes/no</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Enable SSH1 agent compatibility</td>
<td></td>
<td></td>
</tr>
<tr>
<td>✓</td>
<td></td>
<td>Ssh1Compatibility</td>
<td>Yes/no</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Enable SSH1 compatibility</td>
<td></td>
<td></td>
</tr>
<tr>
<td>✓</td>
<td></td>
<td>Ssh1Path</td>
<td>Filename</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Path to ssh1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>✓</td>
<td></td>
<td>SshSignerPath</td>
<td>Filename</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Path to ssh-signer2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 2.5 ssh Options

<table>
<thead>
<tr>
<th>SSH1</th>
<th>SSH2</th>
<th>OpenSSH</th>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>-2</td>
<td>Use SSH-2 protocol only</td>
</tr>
<tr>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>-4</td>
<td>Use IPv4 addresses only</td>
</tr>
<tr>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>-6</td>
<td>Use IPv6 addresses only</td>
</tr>
<tr>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>-8</td>
<td>No effect; simply passed along to <code>rsh</code> on fallback; signifies 8-bit clean connection</td>
</tr>
<tr>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>-a</td>
<td>Disable agent forwarding</td>
</tr>
<tr>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>+a</td>
<td>Enable agent forwarding</td>
</tr>
<tr>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>-A</td>
<td>Enable agent forwarding</td>
</tr>
<tr>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>-c cipher</td>
<td>Select encryption cipher</td>
</tr>
<tr>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>-C</td>
<td>Enable compression</td>
</tr>
<tr>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>-c</td>
<td>Disable compression</td>
</tr>
<tr>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>+C</td>
<td>Enable compression</td>
</tr>
<tr>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>-d level</td>
<td>Enable debug messages</td>
</tr>
<tr>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>-d &quot;module=level&quot;</td>
<td>Enable debug messages per module</td>
</tr>
<tr>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>-e character</td>
<td>Set escape character (<code>^</code> = Ctrl key)</td>
</tr>
<tr>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>-f</td>
<td>Fork into background</td>
</tr>
<tr>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>-F filename</td>
<td>Use other configuration file</td>
</tr>
<tr>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>-g</td>
<td>Gateway locally forwarded ports</td>
</tr>
<tr>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>-h</td>
<td>Print help message</td>
</tr>
<tr>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>-i filename</td>
<td>Select identity file</td>
</tr>
<tr>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>-k</td>
<td>Disable Kerberos ticket forwarding</td>
</tr>
<tr>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>-l username</td>
<td>Remote username</td>
</tr>
<tr>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>-L port1:port2:port2</td>
<td>Local port forwarding</td>
</tr>
<tr>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>-m algorithm</td>
<td>Select MAC algorithm</td>
</tr>
<tr>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>-n</td>
<td>Redirect stdin from <code>/dev/null</code></td>
</tr>
<tr>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>-N</td>
<td>Execute no remote command</td>
</tr>
<tr>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>![checkmark]</td>
<td>-o &quot;keyword value&quot;</td>
<td>Set configuration keyword</td>
</tr>
<tr>
<td>Option</td>
<td>Meaning</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------------</td>
<td>----------------------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>-p port</code></td>
<td>Select TCP port number</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>-P</code></td>
<td>Use nonprivileged port</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>-q</code></td>
<td>Quiet mode</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>-R port1:host2:port2</code></td>
<td>Remote port forwarding</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>-s subsystem</code></td>
<td>Invoke remote subsystem</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>-S</code></td>
<td>No session channel</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>-t</code></td>
<td>Allocate tty</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>-T</code></td>
<td>Don't allocate tty</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>-v</code></td>
<td>Verbose mode</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>-V</code></td>
<td>Print version number</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>-x</code></td>
<td>Disable X forwarding</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>+x</code></td>
<td>Enable X forwarding</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>-X</code></td>
<td>Enable X forwarding</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 2.6 scp Options

<table>
<thead>
<tr>
<th>SSH1</th>
<th>SSH2</th>
<th>OpenSSH</th>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td><code>-l</code></td>
<td>Enable <code>scp1</code> compatibility</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td></td>
<td><code>-4</code></td>
<td>Use IPv4 addresses only</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td></td>
<td><code>-6</code></td>
<td>Use IPv6 addresses only</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td><code>-a</code></td>
<td>No file-by-file statistics</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td><code>-A</code></td>
<td>Print file-by-file statistics</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td><code>-B</code></td>
<td>Disable prompting</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td><code>-c cipher</code></td>
<td>Select encryption cipher</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td><code>-d</code></td>
<td>Require target to be a directory when copying a single file</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td><code>-D &quot;module=level&quot;</code></td>
<td>Enable debug messages per module</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td><code>-f</code></td>
<td>Specify copy FROM (internal use)</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td></td>
<td><code>-h</code></td>
<td>Print help message</td>
</tr>
<tr>
<td>✓</td>
<td>N</td>
<td>✓</td>
<td><code>-i filename</code></td>
<td>Select identity file</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td></td>
<td><code>-L</code></td>
<td>Use nonprivileged port</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td></td>
<td><code>-n</code></td>
<td>Print actions, but don't copy</td>
</tr>
<tr>
<td>✓</td>
<td>N</td>
<td>✓</td>
<td><code>-o &quot;keyword value&quot;</code></td>
<td>Set configuration keyword</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td><code>-p</code></td>
<td>Preserve file attributes</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td><code>-P port</code></td>
<td>Select TCP port number</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td><code>-q</code></td>
<td>Don't print statistics</td>
</tr>
<tr>
<td>✓</td>
<td></td>
<td></td>
<td><code>-q</code></td>
<td>Quiet mode</td>
</tr>
<tr>
<td>✓</td>
<td></td>
<td></td>
<td><code>-Q</code></td>
<td>Print statistics</td>
</tr>
<tr>
<td>✓</td>
<td></td>
<td></td>
<td><code>-Q</code></td>
<td>Don't print statistics</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td><code>-r</code></td>
<td>Recursive copy</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td><code>-S filename</code></td>
<td>Path to <code>ssh</code> executable</td>
</tr>
</tbody>
</table>
2.7 ssh-keygen Options

<table>
<thead>
<tr>
<th>SSH1</th>
<th>SSH2</th>
<th>OpenSSH</th>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-t</td>
<td>Specify copy TO (internal use)</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td></td>
<td>-u</td>
<td>Remove original file</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-v</td>
<td>Verbose mode</td>
</tr>
<tr>
<td>✓</td>
<td></td>
<td></td>
<td>-V</td>
<td>Print version number</td>
</tr>
</tbody>
</table>

### 2.7.1 ssh-keygen Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-l filename</td>
<td>Convert SSH1 key file to SSH2</td>
</tr>
<tr>
<td>-b bits</td>
<td># of bits in generated key</td>
</tr>
<tr>
<td>-B positive_integer</td>
<td>Specify numeric base for displaying key</td>
</tr>
<tr>
<td>-c</td>
<td>Change comment (with -C)</td>
</tr>
<tr>
<td>-c comment</td>
<td>Change comment</td>
</tr>
<tr>
<td>-C comment</td>
<td>Specify new comment (with -c)</td>
</tr>
<tr>
<td>-d</td>
<td>Generate DSA key</td>
</tr>
<tr>
<td>-D filename</td>
<td>Derive public key from private key file</td>
</tr>
<tr>
<td>-e filename</td>
<td>Edit key file interactively</td>
</tr>
<tr>
<td>-f filename</td>
<td>Output filename</td>
</tr>
<tr>
<td>-F filename</td>
<td>Print fingerprint of public key</td>
</tr>
<tr>
<td>-h</td>
<td>Print help and exit</td>
</tr>
<tr>
<td>-l</td>
<td>Print fingerprint of public key</td>
</tr>
<tr>
<td>-N passphrase</td>
<td>Specify new passphrase</td>
</tr>
<tr>
<td>-o filename</td>
<td>Output filename</td>
</tr>
<tr>
<td>-p</td>
<td>Change passphrase (with -P and -N)</td>
</tr>
<tr>
<td>-p passphrase</td>
<td>Change passphrase</td>
</tr>
<tr>
<td>-P passphrase</td>
<td>Specify old passphrase (with -p)</td>
</tr>
<tr>
<td>-P</td>
<td>Use empty passphrase</td>
</tr>
<tr>
<td>-q</td>
<td>Quiet: suppress progress indicator</td>
</tr>
<tr>
<td>-r</td>
<td>Stir in data from random pool</td>
</tr>
<tr>
<td>-R</td>
<td>Detect RSA (exit code 0/1)</td>
</tr>
<tr>
<td>-t algorithm</td>
<td>Select key-generation algorithm</td>
</tr>
<tr>
<td>-u</td>
<td>Change encryption algorithm</td>
</tr>
<tr>
<td>-v</td>
<td>Print version string and exit</td>
</tr>
<tr>
<td>-V</td>
<td>Print version string and exit</td>
</tr>
<tr>
<td>-x</td>
<td>Convert OpenSSH public key to SSH2</td>
</tr>
<tr>
<td>-X</td>
<td>Convert SSH2 public key to OpenSSH</td>
</tr>
<tr>
<td>-y</td>
<td>Derive public key from private key file</td>
</tr>
<tr>
<td>-?</td>
<td>Print help and exit</td>
</tr>
</tbody>
</table>

[1] Undocumented

[2] The output filename is given as the final argument to ssh-keygen2

[3] You may need to escape the question mark in your shell, e.g., -?
2.8 ssh-agent Options

<table>
<thead>
<tr>
<th>SSH1</th>
<th>SSH2</th>
<th>OpenSSH</th>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-l</td>
<td>SSH1 compatibility mode</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-c</td>
<td>Print C shell-style commands</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-k</td>
<td>Kill existing agent</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-s</td>
<td>Print sh-style commands</td>
</tr>
</tbody>
</table>

2.9 ssh-add Options

<table>
<thead>
<tr>
<th>SSH1</th>
<th>SSH2</th>
<th>OpenSSH</th>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-l</td>
<td>Limit SSH1 compatibility</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-d</td>
<td>Unload key</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-D</td>
<td>Unload all keys</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-f step</td>
<td>Limit agent-forwarding hops</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-F host_list</td>
<td>Limit agent-forwarding hosts</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-l</td>
<td>List loaded keys</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-L</td>
<td>List fingerprints of loaded keys</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-L</td>
<td>Lock agent</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-N</td>
<td>PGP keys are identified by name</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-p</td>
<td>Read passphrase from stdin</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-P</td>
<td>PGP keys are identified by fingerprint</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-R filename</td>
<td>Specify PGP keyring file</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-t timeout</td>
<td>Expire key after timeout</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-U</td>
<td>Unlock agent</td>
</tr>
</tbody>
</table>

2.10 Identity and Authorization Files

~/.ssh/authorized_keys (SSH1, OpenSSH/1) and ~/.ssh/authorized_keys2 (OpenSSH/2): use one public key per line, preceded by options.

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>command=&quot;Unix shell command &quot;</td>
<td>Specify a forced command</td>
</tr>
<tr>
<td>environment=&quot;variable=value &quot;</td>
<td>Set environment variable</td>
</tr>
<tr>
<td>from=host_or_ip_address_specification</td>
<td>Limit incoming hosts</td>
</tr>
<tr>
<td>idle-timeout=time</td>
<td>Set idle timeout</td>
</tr>
<tr>
<td>no-agent-forwarding</td>
<td>Disable agent forwarding</td>
</tr>
<tr>
<td>no-port-forwarding</td>
<td>Disable port forwarding</td>
</tr>
</tbody>
</table>
### ~/.ssh2/authorization (SSH2)

Use one keyword/value pair per line.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command Unix_command</td>
<td>Specify a forced command</td>
</tr>
<tr>
<td>Key filename.pub</td>
<td>Location of public key file</td>
</tr>
<tr>
<td>PgpPublicKeyFile filename</td>
<td>Location of PGP public key file</td>
</tr>
<tr>
<td>PgpKeyFingerprint fingerprint</td>
<td>Select PGP key by fingerprint</td>
</tr>
<tr>
<td>PgpKeyId id</td>
<td>Select PGP key by ID</td>
</tr>
<tr>
<td>PgpKeyName name</td>
<td>Select PGP key by name</td>
</tr>
</tbody>
</table>

### ~/.ssh2/identification (SSH2)

Use one keyword/value pair per line.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>IdKey filename</td>
<td>Location of private key file</td>
</tr>
<tr>
<td>IdPgpKeyFingerprint fingerprint</td>
<td>Select PGP key by fingerprint</td>
</tr>
<tr>
<td>IdPgpKeyId id</td>
<td>Select PGP key by ID</td>
</tr>
<tr>
<td>IdPgpKeyName name</td>
<td>Select PGP key by name</td>
</tr>
<tr>
<td>PgpSecretKeyFile filename</td>
<td>Location of PGP private key file</td>
</tr>
</tbody>
</table>

### 2.11 Environment Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Set By</th>
<th>In</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSH_AUTH_SOCK</td>
<td>ssh-agent</td>
<td>SSH1, OpenSSH</td>
<td>Path to socket</td>
</tr>
<tr>
<td>SSH2_AUTH_SOCK</td>
<td>ssh-agent</td>
<td>SSH2</td>
<td>Path to socket</td>
</tr>
<tr>
<td>SSH_CLIENT</td>
<td>sshd</td>
<td>SSH1, OpenSSH</td>
<td>Socket info</td>
</tr>
<tr>
<td>SSH2_CLIENT</td>
<td>sshd</td>
<td>SSH2</td>
<td>Socket info</td>
</tr>
<tr>
<td>SSH_ORIGINAL_COMMAND</td>
<td>sshd</td>
<td>SSH1</td>
<td>Client's remote command string</td>
</tr>
<tr>
<td>SSH_SOCKS_SERVER</td>
<td>sshd</td>
<td>SSH2</td>
<td>SOCKS firewall information</td>
</tr>
<tr>
<td>SSH_TTY</td>
<td>sshd</td>
<td>SSH1, OpenSSH</td>
<td>Name of allocated TTY</td>
</tr>
<tr>
<td>SSH2_TTY</td>
<td>sshd</td>
<td>SSH2</td>
<td>Name of allocated TTY</td>
</tr>
</tbody>
</table>
Our look is the result of reader comments, our own experimentation, and feedback from distribution channels. Distinctive covers complement our distinctive approach to technical topics, breathing personality and life into potentially dry subjects.

The animal on the cover of SSH, the Secure Shell: The Definitive Guide is a land snail (Mollusca gastropoda).

A member of the mollusk family, a snail has a soft, moist body that is protected by a hard shell, into which it can retreat when in danger or when in arid or bright conditions. Snails prefer wet weather and, though not nocturnal, will stay out of bright sun. At the front of a snail's long body are two sets of tentacles: its eyes are at the end of one set, and the other set is used for smelling and navigation.

Land snails are hermaphrodites, each having both female and male sex organs, though a snail must mate with another snail in order for fertilization to occur. A snail lays eggs approximately six times a year, with almost 100 eggs each time. Young snails hatch in a month and become adults in two years. A snail's life span is approximately 5-10 years.

Known as a slow mover, a snail moves by muscles on its underside that contract and expand, propelling the snail along at a slow pace. It leaves a wet trail of mucus, which protects the snail from anything sharp it may need to crawl over as it searches for food. The snail's diet of plants, bark, and fruits causes it to be a pest in many parts of the world where it is notorious for destroying crops.

Mary Anne Weeks Mayo was the production editor and copyeditor for SSH, the Secure Shell: The Definitive Guide. Colleen Gorman proofread the book. Rachel Wheeler and Jane Ellin provided quality control. Matt Hutchinson and Lucy Muellner provided production assistance. John Bickelhaupt revised the index.

Ellie Volckhausen designed the cover of this book, based on a series design by Edie Freedman. The cover image is an original engraving from the book Natural History of Animals by Sanborn Tenney and Abby A. Tenney, published by Scribner, Armstrong & Co. in 1873. Emma Colby produced the cover layout with Quark™XPress 4.1 using Adobe's ITC Garamond font.

David Futato and Melanie Wang designed the interior layout based on a series design by Nancy Priest. Mike Sierra implemented the design in FrameMaker 5.5.6. The text and heading fonts are ITC Garamond Light and Garamond Book; the code font is Constant Willison. The illustrations that appear in the book were produced by Robert Romano using Macromedia FreeHand 8 and Adobe Photoshop 5. This colophon was written by Nicole Arigo.

The online edition of this book was created by the Safari production group (John Chodacki, Becki Maisch, and Madeleine Newell) using a set of Frame-to-XML conversion and cleanup tools written and maintained by Erik Ray, Benn Salter, John Chodacki, and Jeff Liggett.