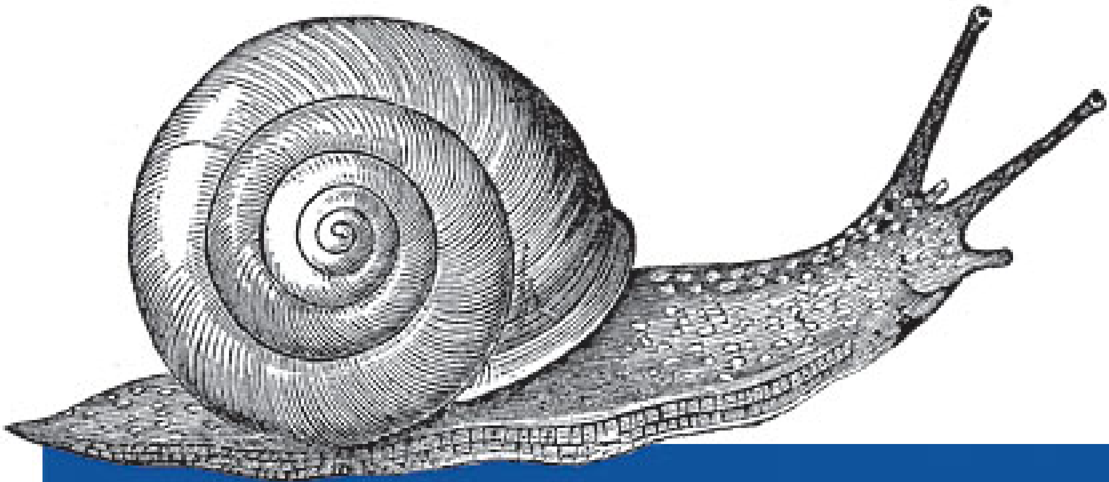
*Securing your Network and Services*

**2nd Edition**
**Covers SSH-2 Protocol**



# SSH
# The Secure Shell

*The Definitive Guide*

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

O'REILLY®

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

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

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

**Printing History:**

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

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

CHAPTER 2

# Basic Client Use

SSH is a simple idea but it has many parts, some of them complex. 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, are covered in later chapters. Our examples in this chapter work with OpenSSH and Tectia on Linux and other Unix-inspired operating systems.

## 2.1 A Running Example

Suppose you're out of town on a business trip and want to access your files, which sit on a Unix machine belonging to your ISP, *shell.isp.com*. A friend at a nearby university agrees to let you log into her Linux account on the machine *local.university.edu*, and then remotely log into yours. For the remote login you could use the *telnet* program, 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 death metal MP3s.) Fortunately, both your friend's 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 Aug 16 19:32:51 2004 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.* You can also provide *user@host* syntax instead of the *–l* option to accomplish the same thing:

```
$ ssh pat@shell.isp.com
```

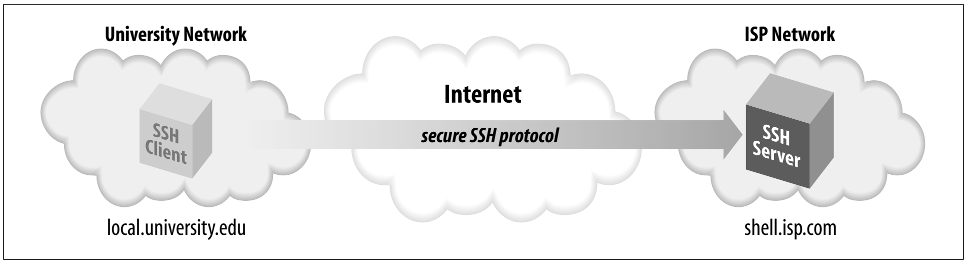


*Figure 2-1. Our example scenario*

On first contact, SSH establishes a secure channel between the client and the server so that 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 everything you type into the SSH application and everything it displays to you from *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 browsing your files, you encounter a PDF file you'd like to print. In order to send the file to a local printer at the university, you

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

must first transfer the file to *local.university.edu*. Once again, you reject as insecure the traditional file-transfer programs, such as *ftp*. 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 *printme.pdf*. 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 for copying files.* It is roughly:

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

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

```
$ scp pat@shell.isp.com:printme.pdf printme.pdf
```

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 default values used. For example, omitting the username and the at sign (pat@) makes *scp* assume that the remote username is the same as the local one.

Like *ssh*, *scp* prompts for your remote password and passes it to the SSH server for verification. If successful, *scp* logs into the pat account on *shell.isp.com*, copies your remote file *printme.pdf* to the local file *printme.pdf*, and logs out of *shell.isp.com*. The local file *printme.pdf* 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 *imprime-moi.pdf*:

```
$ scp pat@shell.isp.com:printme.pdf imprime-moi.pdf
```

The full syntax of *scp* can represent local and remote files in powerful ways, and the program also has numerous command-line options. [7.5]

## 2.3 Adding Complexity to the Example

The preceding example session provided a quick introduction to the most often-used client programs—*ssh* and *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.

* Actually it's modeled after the old *rcp* program for copying files insecurely between machines.

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 *–v* ("verbose") command-line option, for example:

```
$ ssh -v shell.isp.com
```

This causes the client to print lots of information about its progress, often revealing the source of the discrepancy.

## 2.3.1 Known Hosts

The first time an SSH client encounters a new remote machine, it may report that it's never seen the machine before, printing a message like the following:

```
$ ssh -l pat shell.isp.com
The authenticity of host 'shell.isp.com (192.168.0.2)' can't be established.
RSA key fingerprint is 77:a5:69:81:9b:eb:40:76:7b:13:04:a9:6c:f4:9c:5d.
Are you sure you want to continue connecting (yes/no)?
```

Assuming you respond yes (the most common response), the client continues:

```
Warning: Permanently added 'shell.isp.com,192.168.0.2' (RSA) to the list of known
hosts.
```

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. [3.9.4] Unless you

[*] Depending on your client configuration, *ssh* might print a different message and automatically accept or reject the connection. [7.4.3.1]

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. [3.4.3.6] 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 systemwide 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 verifying 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: REMOTE 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 RSA host key has just been changed.
The fingerprint for the RSA key sent by the remote host is
77:a5:69:81:9b:eb:40:76:7b:13:04:a9:6c:f4:9c:5d.
Please contact your system administrator.
Add correct host key in /home/smith/.ssh/known_hosts to get rid of this message.
Offending key in /home/smith/.ssh/known_hosts:36
```

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 key may have legitimately changed for some reason. In some cases, even after reading this book, you won't know the cause of these messages. If you need assistance, contact your system administrator or a knowledgeable friend, 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. [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 *printme.pdf*. 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.* We cover these in detail in a later chapter. [7.4.6.8]

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*. [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. [3.4.2.4] (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:

- 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. [2.4.2]
- You need to install your public key on an SSH server machine. [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 to produce either a DSA or RSA key. The OpenSSH version of *ssh-keygen* requires you to specify the key type with the *–t* option (there is no default):

```
$ ssh-keygen -t dsa
Generating public/private dsa key pair.
Enter file in which to save the key (/home/dbarrett/.ssh/id_dsa): press ENTER
Enter passphrase (empty for no passphrase): ********
Enter same passphrase again: ********
Your identification has been saved in /home/pat/.ssh/id_dsa.
Your public key has been saved in /home/pat/.ssh/id_dsa.pub.
The key fingerprint is:
14:ba:06:98:a8:98:ad:27:b5:ce:55:85:ec:64:37:19 pat@shell.isp.com
```

On Tectia systems, *ssh-keygen* produces a DSA key by default, and also accepts the *–t* option:

```
$ ssh-keygen
Generating 2048-bit dsa key pair
   1 ..oOo.oOo.oO
   2 o.oOo.oOo.oO
   3 o.oOo.oOo.oO
      The program displays a "ripple" pattern to indicate progress; the characters are actually
overwritten on a single line
  28 o.oOo.oOo.oO
Key generated.
2048-bit dsa, pat@shell.isp.com, Wed Jan 12 2005 20:22:21 -0500
Passphrase : **************
Again      : **************
Private key saved to /home/pat/.ssh2/id_dsa_2048_a
Public key saved to /home/pat/.ssh2/id_dsa_2048_a.pub
```

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 or wiggle your mouse around. *ssh-keygen* uses the timings of your keystrokes to initialize its internal random-number generator. On a 3.2 GHz Pentium 4 system running Linux, a 1024-bit RSA key generates in less than one second; if your hardware is slower or heavily loaded, generation could take minutes. It can also take longer if the process runs out of random bits and *ssh-keygen* waits to collect more.

*ssh-keygen* then creates your local SSH directory (*~/.ssh* for OpenSSH or *~/.ssh2* for Tectia) 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 *id_dsa* and *id_dsa.pub* (OpenSSH) or *id_dsa_2048_a* and *id_dsa_2048_a.pub* (Tectia). SSH clients consider these to be your default identity for authentication purposes.

Never reveal your private key and passphrase to anyone. They are just as sensitive as your login password. Anyone possessing them can impersonate 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. [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: *~/.ssh/authorized_keys* for OpenSSH or *~/.ssh2/authorization* for Tectia.

#### 2.4.3.1 Instructions for OpenSSH

Create or edit the remote file *~/.ssh/authorized_keys* and append your public key—i.e., the contents of the *id_dsa.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 artifacts; if they were actually in the file, it would be incorrectly formatted and wouldn't work:

```
ssh-dss AAAAB3NzaC1kc3MAAACBAMCiL15WEI+OdFJZ9InMSh4PAZ3eFO7YJBFZ6ybl7ld+807z/
jnXGghYVuvKbHdNlRYWidhdFWtDW3l5v8Ce7nyYhcQU7x+j4JeUf7qmLmQxluOv+O5rlg7L5U2RuW94yt1BGj
+xk7vzLwOhKHE/+YFVz52sFNazoYXqPnm1pRPRAAAAFQDGjroMj+ML= jones@client2.com
```

```
ssh-rsa
AAAAB3NzaC1yc2EAAAABIwAAAIEAvpB4lUbAaEbh9u6HLig7amsfywD4fqSZq2ikACIUBn3GyRPfeF93l/
weQh702ofXbDydZAKMcDvBJqRhUotQUwqV6HJxqoqPDlPGUUyo8RDIkLUIPRyqypZxmK9aCXokFiHoGCXfQ9i
mUP/w/jfqb9ByDtG97tUJF6nFMP5WzhM= smith@client.net
```

The first entry is a DSA key and the second is RSA. [8.2.1]

### 2.4.3.2 Instructions for Tectia

For Tectia 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_2048_a
```

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

```
Key id_dsa_2048_a.pub
```

Finally, copy *id_dsa_2048_a.pub* from your local machine to the remote Tectia 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:*

```
# OpenSSH
$ chmod 755 ~/.ssh
$ chmod 644 ~/.ssh/authorized_keys

# Tectia
$ chmod 755 ~/.ssh2
$ chmod 644 ~/.ssh2/id_dsa_2048_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. [5.3.2.1]

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

```
$ ssh -l pat shell.isp.com
Enter passphrase for key '/home/you/.ssh/id_dsa': ************
Last login: Mon Aug 16 19:44:21 2004 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.

* We make files world-readable and directories world-searchable, to avoid NFS problems. [10.7.2] But if `StrictModes` is enabled in the server, you'll need to make these permissions more restrictive. [5.3.2.1]

## Installing OpenSSH Keys with ssh-copy-id

OpenSSH includes a program, *ssh-copy-id*, that installs a public key automatically on a remote server with a single command, placing it into *~/.ssh/authorized_keys*:

```
ssh-copy-id -i key_file [user@]server_name
```

For example, to install the key *mykey* in the dulaney account on *server.example.com*:

```
$ ssh-copy-id -i mykey dulaney@server.example.com
```

You don't need to list the *.pub* extension of the key file; or more specifically, you can provide either the private or public-key file, and the public key is copied to the remote server.

In order for the copy to take place, you'll need an account on the remote machine, of course, and you'll need to authenticate somehow. If you've never set up public-key authentication on *server.example.com* before, you'll be prompted for your login password.

*ssh-copy-id* is convenient, but it has some subtle issues:

- If you have no *authorized_keys* file on the remote machine, *ssh-copy-id* creates one containing your new key; otherwise, it appends the new key.
- If you do already have a remote *authorized_keys* file, and it does not end with a newline character, *ssh-copy-id* blindly appends your new key onto the last public key in the file, with no newline between them. This effectively corrupts the last two keys in *authorized_keys*. Moral: always make sure *authorized_keys* ends with a newline. (This is easy to overlook, especially when running OpenSSH on Windows. [14.4])
- The syntax of *ssh-copy-id* is similar to that of *scp*, the secure copy program, but there's an important difference: *scp* follows the hostname of the remote machine with a colon. Don't use a colon with *ssh-copy-id* or you'll get an error message, "Name or service not known," as the hostname lookup fails.

Before you use *ssh-copy-id* to simplify or hide the details of public-key authentication, we recommend that you understand how to set it up manually. This point is often true of security-related software: you should know how and why it works.

Note the similarity to the earlier example with password authentication. [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. [2.4.1]

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

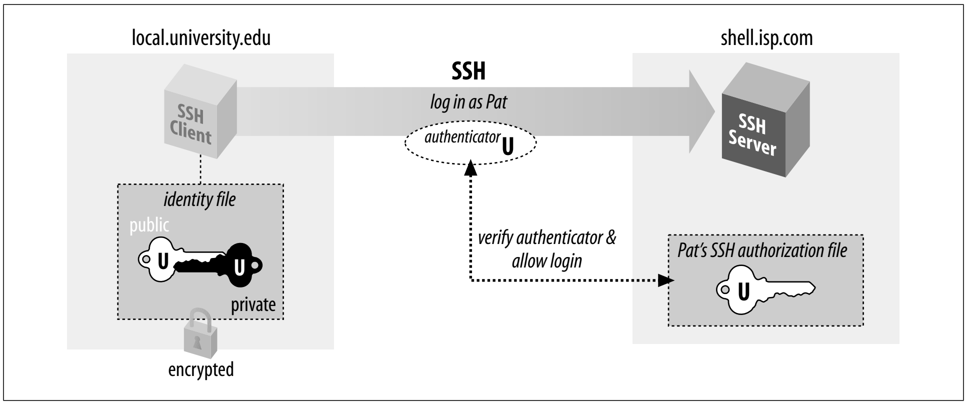


*Figure 2-2. Public-key authentication*

- 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, *id_dsa* and *id_dsa.pub*, and copied *id_dsa.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 *id_dsa* and *id_dsa.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. [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 OpenSSH [6.2.1] and *–e* for Tectia [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 both OpenSSH and Tectia 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 a local socket or named pipe whose filename is stored in an environment variable, so all clients (and all other processes) within your login session have access to the agent. [6.3.4] 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
Enter passphrase for /home/you/.ssh/id_dsa: ********
Identity added: /home/you/.ssh/id_dsa (/home/you/.ssh/id_dsa)
```

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

*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.

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

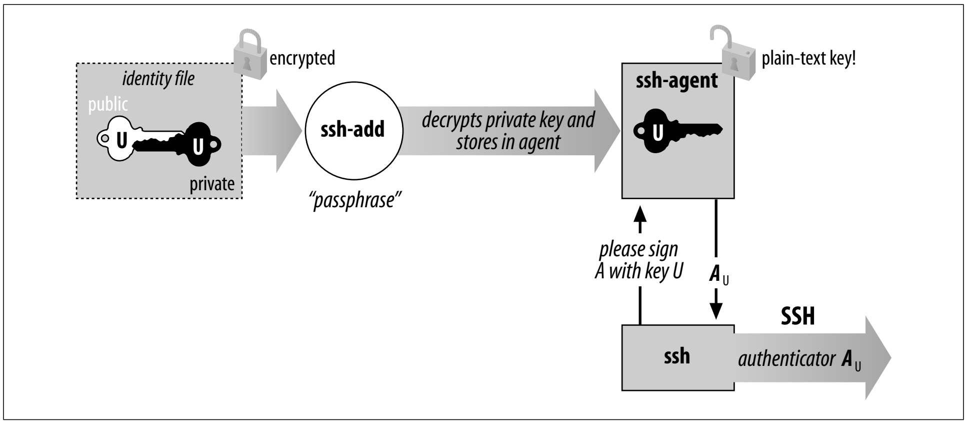


*Figure 2-3. How the SSH agent works*

*ssh-add* has further capabilities and can operate with multiple identity files. [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 in memory:

```
$ ssh-add -d name-of-key-file
```

or delete all keys from the agent in memory:

```
$ 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-agent* can be "locked" by *ssh-add*, to protect the agent from unauthorized users. [6.3.3]

## 2.5.1 Agents and Automation

Suppose you have a batch script that runs *ssh* to launch remote processes. If the script runs *ssh* many times, it prompts for your passphrase repeatedly, which is inconvenient for automation (not to mention annoying and error-prone). If you run an agent, however, your script can run without a single passphrase prompt. [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:printme.pdf imprime-moi.pdf
```

In fact, *scp* can copy a file from the 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:printme.pdf psmith@other.host.net:imprime-moi.pdf
```

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 run into the following problem:

```
$ scp pat@shell.isp.com:printme.pdf psmith@other.host.net:imprime-moi.pdf
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 and 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 host. SSH provides *agent forwarding* [6.3.5] to address this problem.

When agent forwarding is turned on,* 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

* It is on by default in Tectia, but off in OpenSSH.

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.

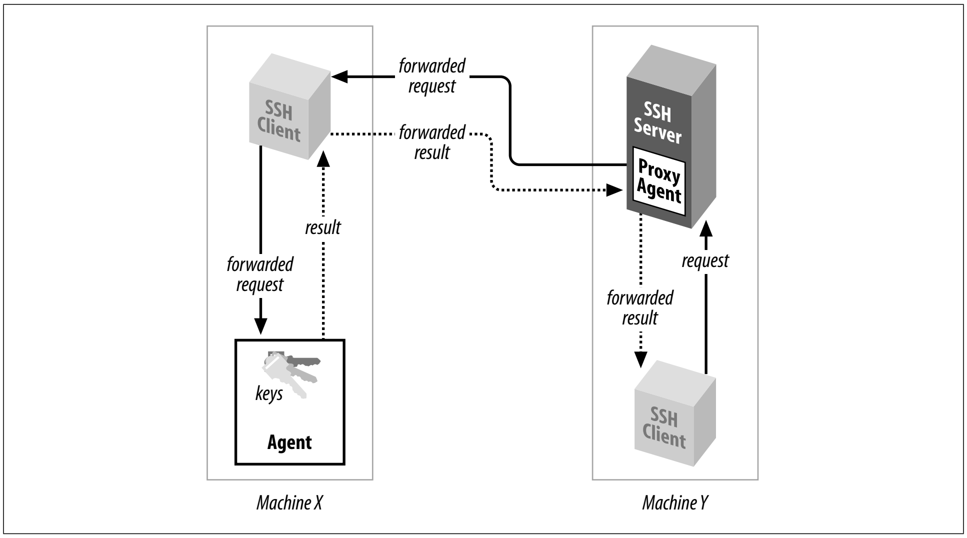


*Figure 2-4. How agent forwarding works*

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:printme.pdf psmith@other.host.net:imprime-moi.pdf
   ```

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.

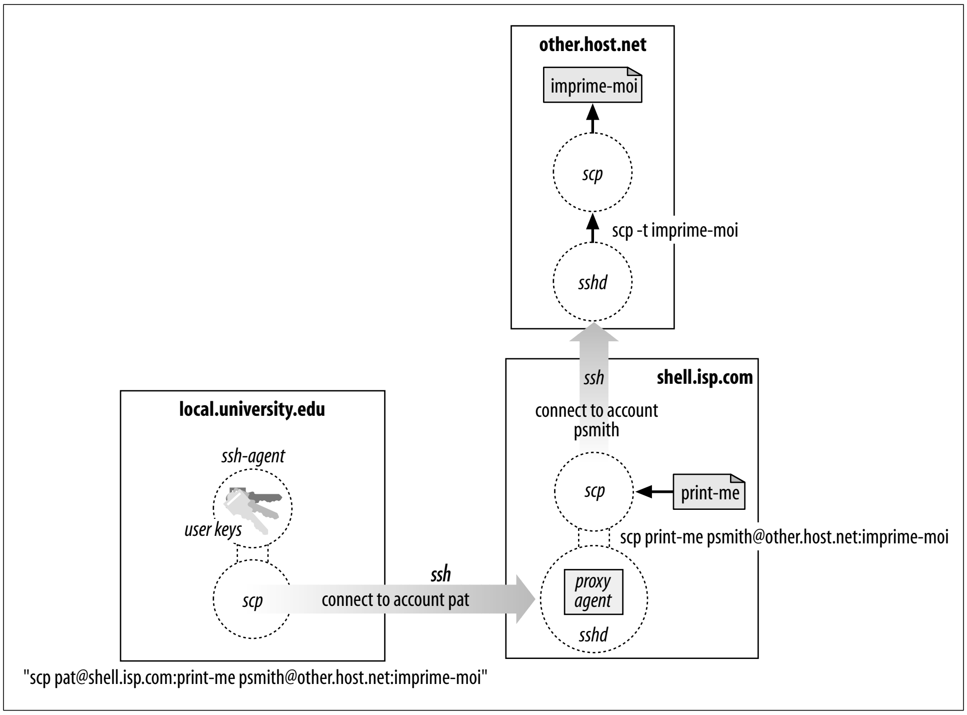


*Figure 2-5. Third-party scp with agent forwarding*

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 agent forwarding doesn't send your private key to the remote host: it just relays authentication requests back to the first host for processing. Therefore, you don't have to copy your private key to other machines.

## 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 trade-offs. 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 [2.5] [6.3]
- Hostbased authentication [3.4.3.6]
- Kerberos authentication [11.4]

> Another way to achieve passwordless 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 discuss their relative merits and security issues later. [11.1]

## 2.7 Miscellaneous Clients

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

- *sftp*, an *ftp*-like client
- *slogin*, a link to *ssh*, analogous to the *rlogin* program

### 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.[*] *sftp* is a separate file-transfer tool layered on top of SSH. The OpenSSH *sftp* can run over either SSH-1 or SSH-2, whereas the Tectia version runs over SSH-2 only due to implementation details.

*sftp* has several advantages:

- 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 might need to run an agent, however, since programs that normally invoke *ftp* might not recognize the *sftp* passphrase prompt, or they might expect you to have suppressed FTP's password prompt (using a *.netrc* file, for example).

[*] Due to the nature of the FTP protocol, FTP clients are difficult to secure using SSH port forwarding. It is possible, however. [11.2]

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).[*]
- Wildcards for matching filenames. OpenSSH uses the same "globbing" syntax that is supported by most common shells, while Tectia uses an extended regular expression syntax described in Appendix B.
- Several useful command-line options:

  *–b filename (OpenSSH)*
  *–B filename (Tectia)*
  : Read commands from the given file instead of the terminal.

  *–S path*
  : Locate the *ssh* program using the given path.

  *–v*
  : Print verbose messages as the program runs.

  *–V (OpenSSH)*
  : Print the program version number and exit.

In addition, many of the command-line options for ssh can also be used for *sftp*.

The OpenSSH version of *sftp* supports only the binary transfer mode of standard FTP, in which files are transferred without modification. Tectia's *sftp* also supports ASCII transfer mode, which translates end-of-line characters between systems that might use different conventions, e.g., carriage return plus newline for Windows, newline (only) for Unix, or carriage return (only) for Macintosh.

### 2.7.2 slogin

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

## 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* and *sftp* for file transfer. Authentication to the remote host can be accomplished

[*] OpenSSH 4.0 and higher.

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.