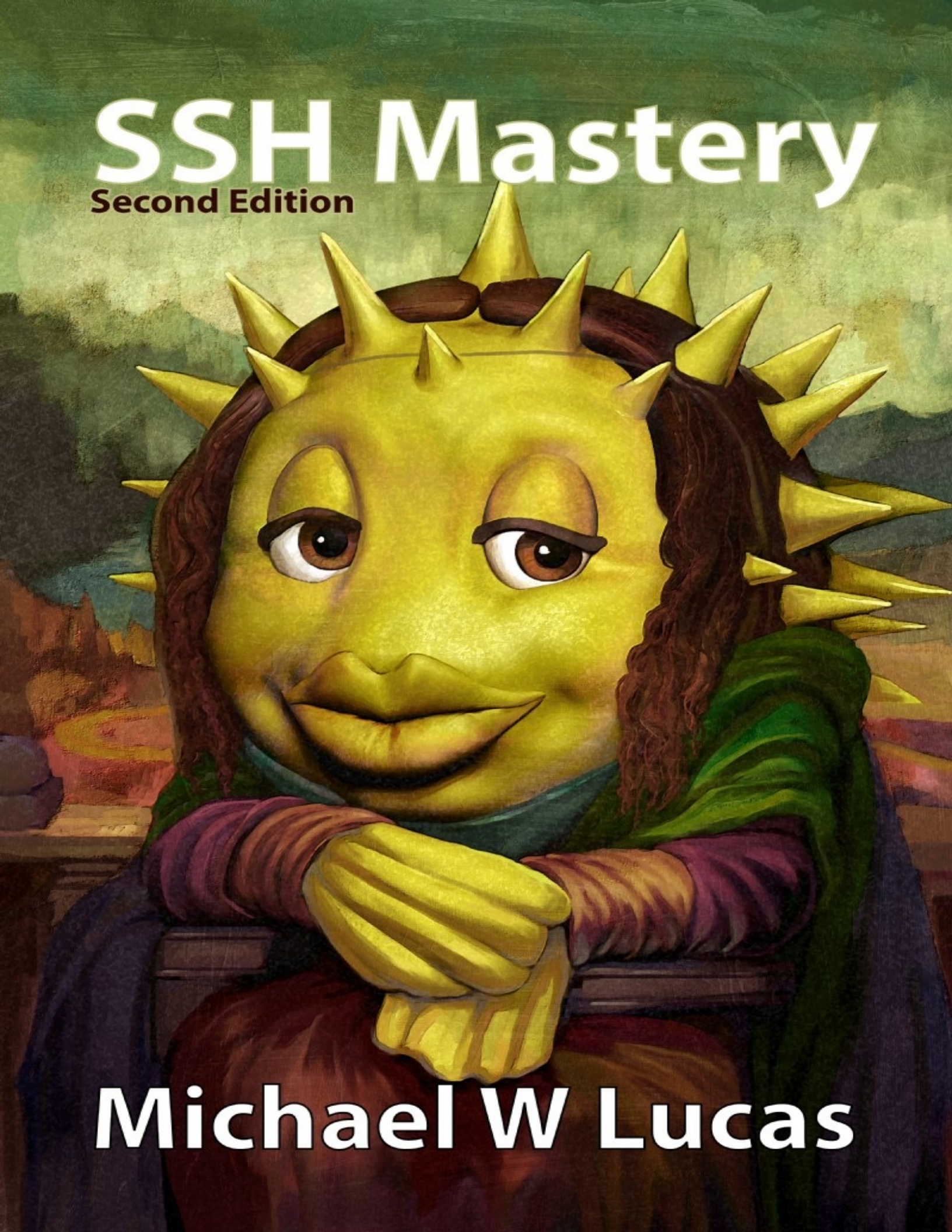
SSH Mastery

Second Edition



Michael W Lucas

Table of Contents

Acknowledgements

Thanks go first to the fine folks who wrote OpenSSH and PuTTY. These people literally changed the world for the better by creating and supporting their software. I must notably thank OpenSSH ringleader Damien Miller, for taking the time to point me in the right direction when I had a dumb question.

I must also thank my technical reviewers: Bill Allaire, Jim Allen, Tim Enders, Marie Helene Kvello-Aune, Kurt Mosiejczuk, Mike O'Connor, Bernard Spil, Loganaden Velvindron (from hackers.mu), and Markus Waldeck. Any errors that appear in this book crept in despite the efforts of these fine folks.

To the people who offer me ongoing support via Patreon (https://www.patreon.com/mwlucas), my gratitude. A whole passel of them got a copy of this book as thanks.

Writing this book would have been impossible without the source code for all the software involved.

This is for Liz.

Chapter 0: Introduction

Over the last 15 years, OpenSSH (http://www.OpenSSH.com) has become the standard tool for remote management of UNIX-like systems and many network devices. Most systems administrators use only the bare minimum OpenSSH functionality necessary to get a command line, however. OpenSSH has many powerful features that will make systems management easier if you take the time to understand them. You'll find information and tutorials about OpenSSH all over the Internet. Some of them are poorly written, or only applicable to narrow scenarios. Many are well written, but are ten years old and cover problems solved by a software update nine years ago. If you have a few spare days, and know the questions to ask, you can sift through the dross and find effective, current tutorials.

This task-oriented book will save you that effort and time, freeing you up to prepare for the next version of Castle Wolfenstein. I assume that you are using fairly recent versions of OpenSSH and PuTTY, and I disregard edge cases such as "my twenty-year-old router only supports SSH version 1." If you found this book, chances are you're capable of searching the Internet to answer very specific questions. I won't discuss building OpenSSH from source, or how to install the OpenSSH server on fifty different platforms. If you're a systems administrator, you know where to find that information. If you are a system user, your system administrator should install and configure the OpenSSH server for you, but mastering the client programs will help you work more quickly and effectively.

Who Should Read This Book?

Everyone who manages a UNIX-like system must understand SSH. OpenSSH is the most commonly deployed SSH implementation. Unless you are specifically using a different SSH implementation, read this book.

People who are not systems administrators, but who must connect to a server over SSH, will also find this book helpful. While you can learn the basics of SSH in five minutes, proper SSH use will make your job easier and faster. You can skip the sections on server configuration if you wish, although it's always good to know what your system administrator *can* actually do as opposed to what they *feel like* doing.

SSH Components

Secure shell (SSH) is a protocol for creating an encrypted communications channel between two networked hosts. SSH protects data passing between two machines so that other people cannot eavesdrop on it. Tatu Ylönen created the

initial protocol and implementation in 1995, designing it to replace insecure protocols such as telnet, RSH, and rlogin. With the release of OpenSSH in 1999, SSH rapidly became the standard method for managing hosts. Today, many different software packages rely on the SSH protocol for encrypted and well-authenticated transport of data across private, public, and hostile networks.

OpenSSH

OpenSSH is the most widely deployed implementation of the SSH protocol. It started as an offshoot of a freely licensed version of the original SSH software, but has been heavily rewritten, expanded, and updated. OpenSSH is developed as part of the OpenBSD Project, a community known for writing secure software. OpenSSH is the standard SSH implementation in the Linux and BSD world, and is also used in products from large companies such as HP, Cisco, Oracle, Novell, Juniper, IBM, and so on.

OpenSSH comes in two versions, *OpenBSD* and *Portable OpenSSH*. OpenSSH's main development happens as part of OpenBSD. They hold OpenSSH to the same standards of simple, secure code as they do the rest of OpenBSD. This version of OpenSSH is small and secure, but only supports OpenBSD. The OpenSSH Portability Team takes the OpenBSD version and adds the glue necessary to make OpenSSH work on other operating systems, creating Portable OpenSSH. Not only do different operating systems use different compilers, libraries, and so on, they have different authentication systems. The Portable OpenSSH team needs to account for all of these differences on every platform. They do their best to hide this complexity, so you don't have to worry about it. This book applies to both versions.

Any operating system probably comes with OpenSSH, or the operating system vendor provides a package. Even Microsoft offers an OpenSSH package in their Linux layer, and a beta of a native port has recently escaped as an optional Windows component. If your operating system doesn't provide an OpenSSH package, download the Portable OpenSSH source code from http://www.OpenSSH.com and follow the instructions to build the software.

OpenSSH is available under a BSD–style license. You can use it for any purpose, with no strings attached. You cannot sue the software authors if OpenSSH breaks, and you can't claim you wrote OpenSSH, but you can use it any way you wish, including adding it to your own products. You can charge to install or support OpenSSH, but the software itself is free.

SSH Server

An SSH server listens on the network for incoming SSH requests, authenticates those requests, and provides a system command prompt (or another service that

you configure). The most popular SSH server is OpenSSH's `sshd`.

SSH Clients

Use an SSH client to connect to your remote server or network device. The most popular SSH client for Windows systems is PuTTY. The standard SSH client for Unix-like systems is ssh(1), from OpenSSH. Both are freely available and usable for any purpose, commercial or noncommercial, at no cost.

Microsoft also recently forked OpenSSH to include an SSH client in Windows. It's considered experimental, though, and development is continuing. Experiment with it as you wish; it should work much like OpenSSH. It's also part of Windows' Linux subsystem. If you're using a Windows-native SSH, though, you really want to use PowerShell rather than the traditional terminal.

Once you understand PuTTY and OpenSSH, you'll have the base knowledge to use any secure SSH client.

SSH Protocol Versions

The SSH protocol comes in two versions, SSH-1 (version 1) and SSH-2 (version 2). Always use SSH-2. All modern SSH software defaults to version 2. You will find old embedded devices that still rely on SSH version 1, but SSH-1 is barely more secure than unencrypted telnet.

One person designed SSH-1 for his own needs. It met those needs admirably, and in the 1990s it was a whole bunch better than telnet. As SSH grew more popular, more people examined the protocol and exposed weaknesses in the original design. With today's computing power, SSH-1 is highly vulnerable to attacks. While SSH-1 encrypts your data in transit and prevents casual eavesdropping, an attacker that knows a couple tricks can capture your data, decrypt your data in transit, lull you into thinking that you logged on to the correct machine when you are actually connected to a different host, insert arbitrary text into the data stream, or any combination of these. Attacking an SSH-1 data stream isn't quite a point-and-click process, but intruders do break SSH-1 in the real world.

The appearance of security is worse than no security. Never use SSH version 1.

It might seem harmless to permit SSH-1 for servers or clients that don't support SSH-2. The client and server transparently negotiate the SSH version they will use for a connection however. If either client or server tolerates SSH-1, an intruder can capture your login credentials and all transmitted data. It's fairly straightforward to insert arbitrary text (such as `rm -rf /*`) into an SSH-1 session. This was discovered in 1998, and today's massive computing power has made this attack far easier. SSH-1 sessions can be decoded in real time by programs

such as Ettercap. The incremental improvements to SSH-1, such as SSH 1.3 and 1.5, are vulnerable. SSH servers that offer SSH version 1.99 support SSH version 1 and version 2.

Do not let your SSH clients request SSH-1. Do not let your SSH servers offer SSH-1.

OpenSSH has removed support for SSH-1, so if you have an old embedded device that only speaks SSH-1, you'll need to manage it with PuTTY or, better still, spend a couple dollars to replace that device with something built this millennium.[1]

SSH-2 is the modern standard. The protocol is designed so that vulnerabilities can be quickly addressed as they are discovered. Our constantly-increasing computing power makes today's strong encryption tomorrow's security risk, so SSH-2 is designed so that its algorithms and protocols can be upgraded in place.

Protocols such as SCP and SFTP (Chapter 7) are built atop SSH.

What Isn't In This Book?

This book is meant to familiarize you with SSH, and help you reach a minimum level of competence with OpenSSH and PuTTY. This means eliminating passwords, restricting your SSH services to the minimum necessary privileges, and using SSH as a transport for common management tools. You will be able to easily copy files over SSH, manage server keys with minimal fuss, use digital certificates to permit only approved keys on your network, and a few other tricks.

This book is not intended as a comprehensive SSH tome. It doesn't cover integrating SSH with Kerberos, or SecurID, or hooking your SSH install into Google authenticator, or using your SSH agent as an authentication source for third-party programs. These are all interesting topics, but very platform specific, and might well change before you finish reading this book. Sysadmins interested in authentication options might find my book *PAM Mastery* (Tilted Windmill Press, 2016) useful.

What Is In This Book?

Chapter 0 is this introduction.

Chapter 1, "Encryption and Keys," gives basic information about encryption and how SSH uses it.

Chapter 2, "Common Configuration," covers configuration syntax used throughout the OpenSSH server and client.

Chapter 3, "The OpenSSH Server," discusses configuring the OpenSSH server `sshd`. This chapter orients you on configuring sshd, but more specific examples appear throughout this book.

Chapter 4, “Host Key Verification,” covers a frequently overlooked but vital part of using any SSH client: verifying server keys. This topic is so vital that it needs its own chapter, even before our first discussion of SSH clients.

Chapter 5, “SSH Clients,” discusses two popular SSH clients, OpenSSH’s ssh(1) for Unix-like systems and PuTTY for Windows.

Chapter 6, “Copying Files Over SSH,” covers moving files across the network using SSH as a transport, with the SCP (secure copy) and SFTP (SSH file transfer) protocols.

Chapter 7, “SSH Keys,” walks you through creating a personal key pair (public and private cryptographic key). Key pairs make authentication more secure. When combined with agents they eliminate the need to routinely type passwords but don’t degrade SSH security.

Chapter 8, “X Forwarding,” will teach you how to display graphics over your SSH connections while minimizing risk.

Chapter 9, “Port Forwarding,” covers using SSH as a generic TCP/IP proxy, letting you redirect arbitrary network connections through the network to remote machines.

Chapter 10, “Keeping SSH Sessions Open,” covers ways to keep SSH sessions running despite the firewalls and proxy servers and unreliable ISPs that want to shut them down after minutes or hours.

Chapter 11, “Key Distribution,” tells systems administrators how to automatically distribute host keys and improve security while eliminating the need for users to manually compare host key fingerprints. We also cover issues in distributing user public keys across large cloud systems.

Chapter 12 “Automation,” discusses ways to use SSH as a transport for automated tools and tightly–controlled user tasks, as well as creating single-purpose user keys.

Chapter 13, “OpenSSH VPNs,” demonstrates how to use OpenSSH to create an encrypted tunnel between two sites.

Chapter 14, “Certificate Authorities,” guides you through creating a certificate authority to permit only authorized user keys to log on to your network.

That’s enough blather! Let’s get to work.

---

[1] A few Linux distributions deliberately ship an SSH client that supports SSH-1. That’s on them.

Chapter 1: Encryption, Algorithms, and Keys

OpenSSH encrypts traffic. What does that mean, and how does it work? I give a detailed explanation in my book *PGP & GPG* (No Starch Press, 2006), but here's the brief version.

Encryption transforms readable *plaintext* into unreadable *ciphertext* that attackers cannot understand. *Decryption* reverses the transformation, producing readable text from apparent gibberish. An *encryption algorithm* is the exact method for performing this transformation. Most children discover the code that substitutes numbers for letters, so that A equals one, B equals two, Z equals 26, and so on. This is a simple encryption algorithm. Modern computer-driven encryption algorithms work on chunks of text at a time and perform far more complicated transformations.

Most encryption algorithms use a *key*; a chunk of text, numbers, symbols, or data used to encrypt messages. A key can be chosen by the user or randomly generated. (People habitually choose easily-guessed keys, so OpenSSH doesn't even give users an option to create your own.) The encryption algorithm uses the key to encrypt the text, making it more difficult for an outsider to decrypt. Even if you know the encryption algorithm, you cannot decrypt the message without the secret encryption key.

Think of the encryption algorithm as a type of lock, and the key is a specific key. Locks come in many different types: house doors, bicycles, factories, and so on. Each uses a certain type of key—your door key is probably the wrong shape to fit into any vehicle ignition. But even a key of the proper type won't work in the wrong lock. Your front door key unlocks your front door, and only your front door. Encryption keys work similarly.

Algorithm Types

Encryption algorithms come in two varieties, symmetric and asymmetric.

A *symmetric algorithm* uses the same key for both encryption and decryption. Symmetric algorithms include, but are not limited to, the Advanced Encryption Standard (AES) and ChaCha20, as well as older but now insecure algorithms like 3DES and Blowfish. A child's substitution code is a symmetric algorithm. Once you know that A equals one and so on, you can encrypt and decrypt messages. Symmetric algorithms (more sophisticated than simple substitution) can be very fast and secure, so long as only authorized people have the key. And that's the problem: an outsider who gets the key can read your messages or replace them with his own. You must protect the key. Sending a key unencrypted across the Internet is like standing on the playground shouting, "A is one, B is

two.” Anyone who hears the key can read your private message.

An *asymmetric algorithm* uses different keys for encryption and decryption. You encrypt a message with one key, and then decrypt it with another. This works because the keys are very large numbers, and multiplying very large numbers is much easier than figuring out how to divide them. (There are very good explanations out on the Internet, if you want the details.) Asymmetric encryption became popular only with the wide availability of computers that can handle the very difficult math, and is much, much slower and more computationally expensive than symmetric encryption.

Having two separate keys creates interesting possibilities. Make one key public. Give it away. Broadcast it to the entire world. Keep the other key private, and protected at all costs. Anyone who has the public key can encrypt a message that only someone who knows the private key can read. Someone who has the private key can encrypt a message and send it out into the world. Anyone can use the public key to decrypt that message, but the fact that the public key can decrypt the message assures recipients that the message sender had the private key. This is the basis of *public key encryption*. The public key and its matching private key are called a *key pair*. Again, think of the lock on your front door. The lock itself is public; anyone can touch it. The key is private. You must have both to get into your home. (You can learn more by researching Diffie-Hellman key exchange.)

How SSH Uses Encryption

Symmetric encryption is fast, but offers no way for hosts to securely exchange keys. Asymmetric encryption lets hosts exchange public keys, but it’s slow and computationally expensive. How can you efficiently encrypt the session between two hosts that have never previously communicated?

Every SSH server has a key pair. Whenever a client connects, the server and the client use this key pair to negotiate a temporary key pair shared only between these two hosts. The client and the server both use this temporary key pair to derive a symmetric key that they will use to exchange data during this session, as well as related keys to provide connection integrity. If the session runs for a long time or exchanges a lot of data, the computers will intermittently negotiate a new temporary key pair and new symmetric key. The SSH protocol is more complicated than this, and include safeguards to prevent many different cryptographic attacks, but cryptographic key exchange is the heart of the protocol.

SSH supports many symmetric and asymmetric encryption algorithms. The client and server negotiate mutually agreeable algorithms at every connection.

While OpenSSH offers options to easily change the algorithms supported and its preference for each, don’t! Programmers with more cryptography experience than both of us together arrived at OpenSSH’s encryption preferences after much hard thought, troubleshooting, and suffering. Gossip, rumor, and innuendo might crown Blowfish as the awesome encryption algorithm du jour, but that doesn’t mean you should tweak your OpenSSH server to use that algorithm and no other.

The most common reason people offer for changing the encryption algorithms is to improve speed. SSH’s primary purpose is security, not speed. Do not abandon security to improve speed. You might encounter a device that only speaks older encryption algorithms. We’ll cope with those in Chapter 15, “OpenSSH Scraps.”

Now that you understand how SSH encryption works, leave the encryption settings alone.

Chapter 2: Common Configuration

The OpenSSH client and server share a common configuration syntax. We'll discuss these common elements before delving into the details of either application. Sysadmins familiar with Unix-like systems should have no trouble with OpenSSH configuration.

All system-wide OpenSSH configuration files reside in `/etc/ssh` by default. Some operating systems use an alternate location—for example, OSX uses `/private/etc/ssh` but symlinks `/etc/ssh` there, while FreeBSD's add-on `openssh-portable` package uses `/usr/local/etc/ssh`. Once you find the configuration directory, you'll find a pretty standard set of files.

Default settings for the ssh(1) client appear in `ssh_config`.

The files starting with `ssh_host` and ending in `_key` are the server's private keys. The middle of each file name gives the encryption algorithm—for example, `ssh_host_ecdsa_key` contains the host key that uses the ECDSA algorithm. These files should only be readable by root.

Each private key has a corresponding file with the same name but an added *.pub* at the end. This is the public key for that file. The server will offer the content of these files to any client.

Finally, `sshd_config` contains the server configuration. While you can tweak `sshd` with command-line options, permanent configuration is handled in the configuration file.

Both configuration files consist of a series of keywords, followed by the value that keyword is set to. These values can have any format that makes sense for the configuration target. Here's how OpenSSH sets one common value.

```
Port 22
```

The keyword Port is set to 22. Presumably this makes sense for whatever the Port keyword is intended to represent. We'll get to what that is in Chapter 3, "The OpenSSH Server." You'll also see keywords set to file paths. Here we see the value HostKey set to a file path in the `/etc/ssh` directory.

```
HostKey /etc/ssh/ssh_host_ed25519_key Follow existing examples when setting any keyword.
Always refer back to the man pages if you have trouble.
```

The pound sign (#) indicates a comment. Everything on a line after a comment is ignored. The OpenSSH crew distributes their configuration files with all options set to the default and commented out. While sshd(8) requires a configuration file to start, you can create a valid, useful, and working configuration with `touch sshd_config`. The SSH client and server run just fine with everything at the default setting. The commented-out settings are provided as a convenient reference, that's all.

To change the defaults, remove the pound sign and change the value.

Multiple Values

Some environments need keywords set to multiple values. How you set those values depends on the keyword. Keywords like HostKey and Port can appear multiple times, each with a separate value.

```
Port 22
Port 2222
```

Keywords like Host accept multiple values, separated by a space.

```
Host envy.mwl.io avarice.mwl.io Other keywords, such as Address, expect comma-separated
values.
Address 192.0.2.0/25, 198.51.100.0/24
```

If ssh(1) or sshd(8) complains about a configuration, verify that you're separating multiple entries correctly. This book contains many examples of assigning multiple values, but the OpenSSH manual is always the final word.

Wildcards in OpenSSH Configuration Files

Configuration files for the OpenSSH server and client accept wildcards, called *patterns*. Rather than listing all possible values of a configuration setting, patterns let you say "anything that matches this expression." Wildcards are most often used for Match rules, as discussed in "Conditional Configuration with Match" later this chapter. Patterns let you write configuration statements such as "all hosts in this domain" or "all IP addresses in this network." The two wildcard characters are: ? matches exactly one character * matches zero or more characters For example, I could use a pattern to set the value of the Host keyword to any host in mwl.io.

```
Host *.mwl.io
```

If I used the question mark wildcard, this pattern would match any host with a one-character hostname. Very few environments segregate security domains by the length of the hostname, but if they did, you could use multiple question marks to identify them. This pattern matches **`sloth.mwl.io`** and **`wrath.mwl.io`**, but not **`gluttony.mwl.io`** or **`avarice.mwl.io`**.

```
Host ?????.mwl.io Patterns are also useful for IP addresses. Here I match the hosts
203.0.113.10 through 203.0.113.19.
Address 203.0.113.1?
```

If I use the asterisk wildcard, I can match any IP within a /24 network.

```
Address 203.0.113.*
```

You might use netmasks with IP address ranges, as discussed in Chapter 5.

Negate patterns by putting an exclamation point in front. This pattern matches everything except hosts in mwl.io.

```
Host !*.mwl.io
```

Negation is most useful when combined with a larger entity—that is, to say "Match everything except that one little piece." If I want to match every host in mwl.io except for the customers in the subdomain vermin.mwl.io, I could use

this pattern. Not all keywords support negation; you'll have to try it and see if it works in your environment.

```
Host !.vermin.mwl.io *.mwl.io The lead OpenSSH developer describes negation as "a little
fiddly." I call it "likely to pull a shiv on you." If you need negation, test thoroughly.
```

Conditional Configuration with Match

Your server might need to behave differently depending on the source address or hostname of an incoming connection, or the username. An SSH client might need to use a different username for a particular group of hosts, or to activate X forwarding (Chapter 9) when used on the local network. The Match *sshd_config* keyword lets you establish special configurations for such situations.

Follow each Match statement by a set of conditions that trigger the match, then by a series of configuration statements OpenSSH should apply to connections that meet all of those conditions. We'll see several examples in the next sections.

Before implementing a Match statement, configure OpenSSH for the most common setting. For example, if you are configuring `sshd`, you might want to deny X forwarding to all but select users. Configure `sshd` to deny X forwarding, then use a Match statement to check the username and permit X forwarding to matching users. While we haven't covered X forwarding yet, denying it is a single entry in *sshd_config*.

```
X11Forwarding no In all of the examples below, such an entry appears near the beginning of
sshd_config as a default setting that we'll selectively override.
```

You cannot use Match statements to adjust all possible *ssh_config* and *sshd_config* keywords. Check the manual pages for the complete list of supported keywords.

Matching Users and Groups

The most common situation I encounter is when I want to enable an option for a particular user or group. The User or Group Match terms permit this.

```
X11Forwarding no Match User mwlucas X11Forwarding yes I am always permitted to use X
forwarding, as my awesome psychic powers eliminate all possible security risks.
```

If all of my system administrators share these powers, or if I settle for exterminating sysadmins who empower intruders, I could Match the whole group containing my sysadmins.

```
X11Forwarding no Match Group wheel X11Forwarding yes If you need multiple Match terms,
separate them by commas.
X11Forwarding no Match User mwlucas, jgballard X11Forwarding yes I know when to use X
forwarding. My user claims he does, too. We'll see.
```

Matching Addresses or Hosts

Perhaps you must permit X forwarding, but only from particular networks. You can match on IP addresses.

```
X11Forwarding no Match Address 203.0.113.0/29, 198.51.100.0/24
X11Forwarding yes If you set UseDNS to yes in sshd_config Match will accept hostnames,
with the usual DNS security and availability caveats.
X11Forwarding no Match Host *.mwl.io, *.michaelwlucas.com X11Forwarding yes Double-check
```

that a DNS failure won’t lock you out of your DNS server and prevent you from fixing the problem.

For *ssh_config* only, skip the word Match when using per-host configurations.

```
X11Forwarding no Host avarice
X11Forwarding yes This configuration statement in ssh_config predates the Match syntax.
```

Multiple Match Conditions

You can list multiple Match terms on a single line. Here, I permit a single user to use password authentication if they connect from a certain IP address.

```
Match Address 192.0.2.8 User djm PasswordAuthentication yes The user djm can log in via password, but only from the host at 192.0.2.8.
```

Placing Match Statements

All configuration statements that follow a Match statement belong to that Match statement, until another Match statement appears or until the file ends. This means that Matches must appear at the end of the configuration file. Consider the following snippet of *sshd_config*.

```
…
X11Forwarding no PasswordAuthentication no …
Match Group wheel X11Forwarding yes
Match Address 192.0.2.0/29, 192.0.2.128/27
PasswordAuthentication yes The keywords X11Forwarding and PasswordAuthentication are set to no. When a user in the group wheel logs in, sshd sets the option X11Forwarding to yes for that user. When a user logs in from an IP address in 192.0.2.0/29 or 192.0.2.128/27, the PasswordAuthentication option gets set to yes. If a user in the wheel group logs in from one of those addresses, he gets both options.
```

We’ll demonstrate Match statements for both sshd(8) and ssh(1) throughout this book.

Now let’s talk about the OpenSSH server.

Chapter 3: The OpenSSH Server

The OpenSSH server `sshd` is highly configurable and lets you restrict who may connect to the server, what actions those users can take, and what actions it permits. Every modern Unix-like operating system comes with `sshd` installed as part of the base operating system.

We'll look at some basics of running `sshd`, and proceed to various global configuration options. More specific options get discussed in relevant chapters of this book.

Is sshd Running?

From a client, the simplest way to test if a server is running an accessible SSH daemon is to try to log into the server. While that's great when everything works, a failure to connect means that either the client or server could be busted, or maybe you have a packet filter in the middle. SSH normally runs on TCP port 22. Use netcat(1) to see if you can access the daemon.

```
$ nc -v devio.us 22
Connection to devio.us 22 port [tcp/ssh] succeeded!
SSH-2.0-OpenSSH_7.0
^C
```

When you connect over raw TCP, `sshd` returns a banner giving the SSH protocol version, the SSH server software, and the software version. This host uses SSH protocol 2, provided by OpenSSH version 7.0.

If you don't get something similar perhaps `sshd` isn't running, or maybe you have a packet filter in the way.

From the server, check and see if the `sshd` process is running.

```
$ ps ax | grep sshd
626 - Is 0:00.03 /usr/sbin/sshd 31960 - Is 0:00.38 sshd: mwlucas [priv] (sshd) 44387 - S
0:05.75 sshd: mwlucas@pts/0 (sshd) This host shows three sshd(8) processes. The first, PID
626, shows plain old /usr/sbin/sshd. It's the master process that listens to TCP port 22.
```

The second process, PID 31960, is the privileged process that handles my SSH connection into this host. The third, PID 44387, is the unprivileged child process that handles your login session. OpenSSH improves security through privilege separation, discussed in "Protecting the SSH Server" at the end of this chapter. If someone has deliberately disabled privilege separation and is running `sshd` insecurely, you won't see the unprivileged sessions.[1]

If `sshd` isn't running, enable it through your operating system configuration tool.

Configuring sshd

Most operating systems run `sshd` as a standalone server without any command-line arguments. The usual way to configure `sshd` is through the keywords in `/etc/ssh/sshd_config`. Before you start mucking with changes in that file, though,

you should know how to test and debug them.

OpenSSH makes debugging `sshd` configurations as simple as possible. You must be **`root`** to run `sshd`, debugging or not. The simplest debugging methods are alternate configuration files, alternate ports, and debugging mode.

Alternate Configuration Files and Ports

Suppose you want to edit *`sshd_config`*, but need to be sure that the change works as expected. The `-f` command-line argument tells sshd(8) to use an alternate configuration file.

```
# /usr/sbin/sshd -f sshd_config.test
```

Note that I executed this test configuration using the full path to `sshd`. OpenSSH's sshd re-executes itself when accepting a connection, and it needs the full path to do so. If you don't give the full path, you'll get an error like `"sshd re-exec requires execution with an absolute path."`

Only one `sshd` instance can attach to a particular TCP port. Your test `sshd` process probably won't start because it cannot bind to port 22. You could edit *`sshd_config.test`* to assign your test process another port, but then you have to re-edit the file when moving it to production, and we all know that's exactly the point that will figure prominently in the outage report. Instead, override the configured TCP port and assign a new one with the `-p` command-line argument.

```
# /usr/sbin/sshd -f sshd_config.test -p 2022
```

The test process is now listening on port 2022. (Note that `-p` cannot override a ListenAddress keyword binding `sshd` to a port as well as an address; see "Network Options" later this chapter.) By setting an alternate configuration file and port on the command line you can test your new configuration, approve it, and move it into production, confident that you didn't wreck a file in making the final, untested change. (Not that I've ever broken a system that way, mind you.) In any case, save your original *`sshd_config`*, just in case your change causes problems testing didn't expose.

Remember to kill your test `sshd` process when you finish testing.

Validating sshd_config Changes

Perhaps you want to make a minor change and think you don't need to perform a full test. You can ask sshd(8) to verify the configuration file and all the key files with the `-t` flag.

```
# sshd -t
/etc/ssh/sshd_config: line 112: Bad configuration option: ExposeAuthInfo
/etc/ssh/sshd_config: terminating, 1 bad configuration options Either the version of sshd
installed on this host is too old to support the ExposeAuthInfo keyword, or the operating
system packager deliberately removed the option.
```

Debugging sshd(8)

The `-d` flag tells sshd to run in foreground debugging mode, without detaching from the controlling terminal. In debugging mode, sshd can only handle a single

login request—no, not one request at a time. It processes one login or login attempt, and exits. Don’t do this in production; run it on an alternate port. Debugging displays everything your `sshd` process does, in real time, like so.

```
# /usr/sbin/sshd -p 2022 -d
debug1: sshd version OpenSSH_7.5, OpenSSL 1.0.2l-freebsd 25 May 2017
debug1: private host key #0: ssh-rsa SHA256:N+faE/OyKhlho8MR8Vw3uhdo75aiuhYotnP/gOOe82E
debug1: private host key #1: ecdsa-sha2-nistp256
SHA256:Q1buYGtWowrN1/8g/EaTEMQr+69h+/Pai3xI4LXN0c8
debug1: private host key #2: ssh-ed25519
SHA256:0TCTf0jZUxzu8dahNrLmuKu19T0BkruI4e3mPOjVInE
debug1: rexec_argv[0]='/usr/sbin/sshd'
debug1: rexec_argv[1]='-p'
debug1: rexec_argv[2]='2022'
debug1: rexec_argv[3]='-d'
debug1: Bind to port 2022 on ::.
debug1: Server TCP RWIN socket size: 65536
Server listening on :: port 2022.
debug1: Bind to port 2022 on 0.0.0.0.
debug1: Server TCP RWIN socket size: 65536
Server listening on 0.0.0.0 port 2022.
```

The debug session starts with the identifying information for your version of sshd(8)—in this case, OpenSSH 7.5, built with OpenSSL 1.0.2l, as part of FreeBSD. We then see three private keys being loaded, using RSA2, ECDSA, and ED25519. The daemon parses its arguments and binds to a port.

If the daemon can’t start, it’ll say why, very clearly, right here. You might have to read the manual page or do a few Internet searches to figure out what the error means, but you’ll know the exact problem.

Connect to this server with an SSH client, and you’ll get hundreds of lines of debugging output as the server and client agree upon encryption protocols, the user attempts to authenticate, and various SSH features like X forwarding are negotiated. I won’t walk you through such a session, as the output varies widely depending on the client, the authentication method, and the SSH features requested and offered.

If you have a problem with SSH, run the server in debugging mode, connect with a client, and read the output. Most often, `sshd` will tell you exactly what the problem is.

When you finish debugging, log out of the client. The sshd(8) process will clean up after itself and exit. You can also unceremoniously terminate `sshd` and throw the client out by hitting CTRL-C.

If a single `-d` doesn’t provide enough detail, add multiples to increase verbosity. Running `/usr/sbin/sshd -dd` should quench your curiosity. If not, add more `-d`’s until you are no longer curious.

Configuring sshd(8)

This chapter discusses some generally useful sshd(8) options. Most *`sshd_config`* options appear in the chapter where they’re most useful—that is, options

affecting X forwarding appear in Chapter 8, “X Forwarding,” while certificate options appear in Chapter 14, “Certificate Authorities.”

The version of OpenSSH shipped with your operating system might not support all of the keywords described in this book. I’ve written this based on OpenSSH 7.6. Some operating systems either ship older versions, or deliberately remove functions for their own reasons. If a configuration option doesn’t work on your server, consult your operating system documentation or ask your vendor.[2]

Set Host Keys

The HostKey keyword gives the full path to a file containing a private key. Each supported encryption algorithm uses a separate file.

```
HostKey /usr/local/etc/ssh/ssh_host_rsa_key HostKey /usr/local/etc/ssh/ssh_host_ecdsa_key
HostKey /usr/local/etc/ssh/ssh_host_ed25519_key The default files are named after the type
of key they contain. The file ssh_host_rsa_key contains an RSA key, ssh_host_ed25519_key
is an ED25519 key, and so on. This isn’t mandatory—OpenSSH will figure out what type of
key is in a file and load it if appropriate—but it’s definitely the best practice. Putting
your RSA key in a file named after ED25519 will confuse everyone.
```

Different operating systems handle missing key files differently. BSD-style and Red Hat-based systems automatically create missing key files. Many Linux systems require the sysadmin to manually create missing key files, but integrate key creation into their usual system administration tools. For example, Debian-based systems create missing key files when you run `dpkg-reconfigure openssh-server`.

Chapter 7, “SSH Keys,” covers creating host keys using OpenSSH’s native tools.

Network Options

You can control how sshd(8) uses the network, from the version of IP all the way to the TCP port.

```
Port 22
AddressFamily any
ListenAddress 0.0.0.0
ListenAddress ::
```

The Port keyword controls the TCP port `sshd` uses. Internet standards call for SSH to run on port 22. Some organizations use a different port for SSH in the hope of improving security. Running SSH on an unusual port won’t actually help secure SSH, but it will reduce the number of login attempts from SSH-cracking worms, as discussed in “Protecting the SSH Server” later this chapter. It also lets you escape particularly ineffective firewalls. Override the Port keyword on the command line with `-p`.

AddressFamily refers to the version of TCP/IP sshd uses. To use only IPv4, set this to *inet*. To only use IPv6, set this to *inet6*. The default, *any*, tells sshd to process requests no matter what protocol they arrive over. Some operating

systems patch sshd(8) to support non-TCP/IP protocols such as the Stream Control Transmission Protocol (SCTP).

Many hosts have multiple IP addresses. By default, `sshd` listens for incoming requests on all of them. If you want to limit the IP addresses that `sshd` attaches to, use the ListenAddress keyword. A ListenAddress of 0.0.0.0 means "all IPv4 addresses," while :: means "all IPv6 addresses." (Some operating systems use :: to mean "all IPv4 and IPv6 addresses," because why would they let you turn on a service for IPv6 only?) Each ListenAddress takes a single IP address as an argument, but you can use as many ListenAddress keywords as you need. Explicitly list every IP address that you want the SSH server to accept connections on.

If a host has many IP addresses and you want to block SSH access to just a few of them, you might find blocking traffic with a packet filter easier than using many ListenAddress statements.

You can also use ListenAddress to add an additional port on a particular IP address, by specifying the port in a ListenAddress statement. Consider the following configuration.

```
ListenAddress 0.0.0.0
ListenAddress 192.0.2.8:2222
```

Our first ListenAddress, 0.0.0.0, tells `sshd` to listen to all addresses on this machine. The default Port is 22, so we'll get port 22 on all addresses. That's fine. The second ListenAddress makes `sshd` also listen for connections on port 2222 on the address 192.0.2.8. Each address can have its own ListenAddress statement.

```
ListenAddress 192.0.2.8:2222
ListenAddress 192.0.2.9:25
ListenAddress 192.0.2.10:80
```

Three different addresses, each with a different port. Mind you, having `sshd` listen to the SMTP and HTTP ports is generally unwise, but OpenSSH is not designed to prevent you from doing generally unwise things. If you're stuck behind a naïve firewall that blocks everything but ports 80 and 443, running `sshd` on those ports would let you evade the firewall.[3]

Banners and Login Messages

Many sysadmins want to display a message to the user before they log in. This is called a *banner*. The SSH protocol doesn't require clients to display banners. The server can offer a banner, but you can't guarantee that the user will see it. Both ssh(1) and PuTTY display banners. Set the keyword Banner to the full path of the file.

```
Banner /etc/ssh/banner
```

Be aware that if the banner does work, it might interfere with automated processes run over SSH. In some locations, a banner can serve as a legal notice to intruders. (Mind you, I'm not aware of anyone who's been successfully

prosecuted through use of such banner warnings, but that is the law.) Choose the headache you prefer.

If the user is authenticating with public keys and the client does display the banner, the login will proceed. No human being will see your legal department's finely worded warning about logging into the host until the login is complete.

You can reliably display the system message of the day, `/etc/motd`. This message doesn't appear until after the client has authenticated, though, so it might not meet your needs. The keyword PrintMotd is set to yes by default, but you can turn it off.

```
PrintMotd yes
```

On systems that use Pluggable Authentication Modules (PAM), a PAM module might be responsible for printing `/etc/motd`. If you're having trouble enabling or disabling the display of `/etc/motd`, check your PAM configuration.

Once a user has logged on, `sshd` prints the time of the user's last logon and where they logged in from. To turn this off, set PrintLastLog to no.

```
PrintLastLog yes
```

While it might seem unnecessary, I strongly recommend leaving PrintLastLog on. More than once, users have alerted me to intrusions when they saw that their previous login was from a foreign country or at a ridiculous hour.

Authentication Options

In a default OpenSSH install, a user can try to log in 6 times in 2 minutes in a single SSH session. You should be using public key authentication (Chapter 7, "SSH Keys") almost everywhere, but even users with passwords should be able to incorrectly type their password in twenty seconds. You can change both the timing and the number of attempts.

The LoginGraceTime keyword controls how long `sshd` gives a user to authenticate. If a session connects to `sshd` for this long without successfully authenticating, the connection terminates. You can give a number of seconds (s), minutes (m), or hours (h).

```
LoginGraceTime 2m
```

You can also control how many times a user may attempt to authenticate in a single connection with MaxAuthTries. The default is 6.

```
MaxAuthTries 6
```

After half of a user's permitted attempts in a single session have failed, `sshd` logs further failures. Authentication attempts include both public key authentication and passwords. After MaxAuthTries failures, the user must initiate a new SSH session and try again.

My usual failure procedure is to fail to log in six times, then remember that I have a different username on this machine. When I take my own advice on changing usernames from Chapter 5, "SSH Clients," and install my public key

everywhere as in Chapter 7, “SSH Keys,” this problem goes away.

Verify Login Attempts against DNS

A log message like “Login failed from boss’s computer” makes you sigh. A log message like “Login succeeded from Hacker Haven Nation” should trigger alarm. The owner of an IP address controls the reverse DNS for that address. An intruder who controls the reverse DNS for his IP address can change the apparent hostname to something within your company. For protection against this sort of attack, `sshd` can verify connection attempts against forward DNS entries.

```
UseDNS no
```

When set to `yes`, every time a client connects, `sshd` looks up the host name for the source IP, and then looks up the IP address for the host name. If the DNS names don’t match, `sshd` rejects the connection.

Suppose an intruder controls the reverse DNS for his IP address 192.0.2.99. He gives it a hostname within your organization, such as dhcp12.mwl.io, and connects to your SSH server. Your SSH server asks its DNS server for the IP address for dhcp12.mwl.io. If that DNS entry doesn’t exist, or it points to an IP other than 192.0.2.99, `sshd` rejects the connection.

If DNS fails, `sshd` waits for a full DNS timeout before allowing the connection.

UseDNS requires that your DNS be tidy, coherent, and correct. While I’m in favor of auditing an organization’s DNS entries, performing such audits via UseDNS lacks elegance. DNS checks don’t help if an intruder can poison the server’s DNS cache. If you’re a home user, your ISP probably controls the reverse DNS on your connection. Also, DNS checks can increase system load. If you serve hundreds or thousands of simultaneous SSH users, that load can be substantial. When DNS fails, failed DNS checks will slow down all SSH logins. Finally, many IPv6 sites haven’t configured reverse DNS and won’t for the foreseeable future.

I discourage enabling UseDNS.

System Administration Features

Tell sshd(8) where to stash its process ID file with the PidFile keyword. Don’t do this lightly. Many management tools (foolishly) use the PID file.

```
PidFile /var/run/sshd.pid
```

This file is written before sshd(8) reduces its privileges, so it can be owned by `root`. If you want to disable writing a PID file, set PidFile to `none`.

The sshd(8) process logs via syslogd, defaulting to the AUTH facility and the INFO level. Control these with the SyslogFacility and LogLevel keywords.

```
SyslogFacility Auth
LogLevel INFO
```

The SyslogFacility keyword accepts any syslog facility. Check the documentation for syslogd(8) for a list of facilities.

Not only does syslogd use LogLevel to determine where to send log messages, sshd(8) uses it to determine what to send to syslogd.

A LogLevel of QUIET logs nothing.

LogLevel FATAL logs only when sshd(8) dies.

The ERROR LogLevel reports only problems.

LogLevel INFO logs problems and when people login and logoff.

VERBOSE logs every detail that doesn't violate privacy, including the fingerprints of public keys used to authenticate.

The DEBUG1, DEBUG2, and DEBUG3 LogLevels send enough data to violate user privacy. Debug messages get sent to syslogd. Most default logging systems don't capture this level of detail; you'll need to configure yours to capture all these details. Also, don't send debug data across an open network using traditional unencrypted syslogd.

Changing Encryption Algorithms

You might find the keywords Cipher and Mac in your configuration. They don't appear in the `sshd_config` provided by OpenSSH, but some operating systems add them. These settings allow you to change the encryption methods your server supports.

Don't muck with these settings. You will only hurt yourself.

Certain organizations, most commonly governments, require using only approved encryption algorithms. The most well-known is the United States' FIPS standard. Such organizations have very specific documents mandating how to configure SSH to comply.

How Many Unauthenticated Connections?

OpenSSH avoids the headaches of threaded programming by starting a separate process to handle each incoming connection. A common denial of service attack against hosts running such programs is to start a whole bunch of client connections until the server exhausts all its resources and falls over. OpenSSH avoids this problem with the MaxStartups option.

MaxStartups lets you set a number of simultaneous unauthenticated connections to the SSH daemon. Once this many connections are trying to authenticate, `sshd` won't accept another connection until an existing connection fails or LoginGraceTime expires for an existing unauthenticated connection. A simple value like 10 protects the server, but doesn't let you log in to do something to try to defend against an ongoing attack.

A better choice is to use Random Early Drop (RED), a protocol long used by

network engineers to avoid congestion. A DOS attack isn't exactly network congestion, but it shares a whole bunch of characteristics with network congestion. RED works by setting throttling limits. Once incoming connections exceed a lower limit, `sshd` gives each subsequent incoming connection a chance of being flat-out rejected. The chance of rejecting a connection increases until the number of unauthenticated connections reaches an upper limit, where all connections are rejected. Using RED means that an attacker needs to throw a monstrous amount of resources at an SSH server to guarantee the sysadmin can't get in. It doesn't make the attack any less annoying, but it does give the sysadmin (and legit users) a chance to log in during the attack.

Configure RED for `sshd` by specifying the lower limit, the initial chance of rejecting a connection, and the upper threshold. The default is 10, 30, and 100.

```
MaxStartups 10:30:100
```

This means that `sshd` accepts up to 10 unauthenticated connections simultaneously. The 11th simultaneous unauthenticated connection has a thirty percent chance of being refused. The odds of a connection being refused increase linearly until the upper threshold of 100, where all connections are refused.

Using RED means that if you keep trying to connect during a DOS attack, you'll eventually get a winning ticket and be admitted.

We talk more about defending `sshd` in "Protecting the SSH Server" at the end of this chapter.

Restricting Access by User or Group

Many networked applications rely on user accounts from the underlying operating system. People use an application over a web page or proprietary client, but never actually SSH into the host. If Fred down in shipping needs access to the Enterprise Resource Planning system to print his shipping labels, and the ERP system requires an underlying user account, the host needs an account for Fred. This isn't ideal practice, but it is reality. If you're responsible for such an application, configure the host so that such users cannot log on to the server.

OpenSSH supports user restrictions with the DenyUsers, AllowUsers, DenyGroups, and AllowGroups options. These options take comma-delimited lists of users or groups as arguments, and are processed in that specific order. The first match wins.

A user listed in DenyUsers cannot log in via SSH, even if listed later in AllowUsers or AllowGroups.

A user listed in AllowUsers can log in via SSH, unless explicitly forbidden in DenyUsers.

A user that belongs to a group listed in DenyGroups cannot log in via SSH, unless specifically permitted to by an AllowUsers statement. This lets you make exceptions for a user.

Lastly, as you might guess, a user that belongs to a group listed in AllowGroups can log in via SSH.

Additionally, the presence of an AllowUsers or AllowGroups entry implies that nobody else can log in. The system denies SSH logins to everyone who is not explicitly permitted.

These restrictions work on a first match basis. Statements are processed in order, and when a user matches a rule, the rule applies immediately and processing stops.

Confused? Let's look at some examples. My host has four users: **backup**, **mwlucas**, **pkdick**, and **jgballard**. They are in groups as below.

```
wheel: mwlucas
staff: mwlucas, pkdick, jgballard
support: pkdick, mwlucas
billing: jgballard
```

While these are small groups, the principles apply to groups of any size.

The billing application requires system accounts, but the user doesn't need access via SSH. If I just want to block the user from the billing department from logging in via SSH, I could use DenyUsers.

```
DenyUsers jgballard
```

All users not listed would still have SSH access. When I add another user from that department, though, I must explicitly add them to DenyUsers. I'm better served by blocking access by group.

```
DenyGroups billing
```

With this one statement, I can add a user to the **billing** group and they automatically can't get their money-grubbing mitts on my precious virtual terminals.

The presence of an AllowGroups statement means that only members of that group can log in. On a BSD system, **wheel** is the group for system administrators. Ubuntu does something similar with the **admin** group, but I'm a BSD guy so you get my preferences. To allow only sysadmins to log in via SSH, use AllowGroups.

```
AllowGroups wheel
```

Anyone in the **wheel** group can log in. While I haven't explicitly forbidden anyone else from logging in, the users **backup**, **pkdick** and **jgballard** are not in the wheel group, so they're out.

I'm the only member of the **wheel** group. I could list myself explicitly.

```
AllowUsers mwlucas
```

I do hope to eventually have help, though. When that day comes, I'll have to create an account for my new sysadmin and add them to the AllowUsers

statement on all of my machines. I'll forget one or the other. Use groups whenever possible.

The support team has access to a different host. I have one particular system where a certain person is forbidden to log in. Here I block that user, but permit the group.

```
DenyUsers pkdick
AllowGroups support
```

This demonstrates "first match wins." User `pkdick` is denied immediately, and that decision is final. Other users can proceed to the AllowGroups statement. You might use this setup on, say, a Raspberry Pi's built-in `pi` account.

Some applications, like properly-configured rsync, need accounts with SSH access. This requires a user account with public key authentication (Chapter 7, "SSH Keys"). These accounts can be dangerous. While you can restrict the accounts that the user can run when authenticated with a key, you don't want rsync connections from random hosts, and you don't want a user with shell access able to circumvent restrictions by editing a file he owns. You can use these Allow and Deny options to restrict where users can come from by adding an @ and an IP address after the username.

```
AllowUsers backup@192.0.2.0/24
AllowGroups support
```

Users in the `support` group can log in from anywhere, and the user `backup` can log in from any host with an IP between 192.0.2.0 and 192.0.2.255. All other users are rejected.

With sensible group memberships and thoughtful Allow and Deny options, you can restrict login access almost any way you need. When in doubt, give accounts the least level of privilege that lets users and programs accomplish their required tasks.

Root SSH Access

Sometimes it might seem that you must allow users, sysadmins, or applications to SSH into the system as `root`. In almost all environments, this is a colossally bad idea. When users must log in as a regular user and then change to `root`, the system logs the user's account, providing accountability and attribution. Logging in as `root` destroys that audit trail. Many server programs are initially started by `root`, and the environment changes that make a user account friendly can propagate into those programs' environments, disrupting service.

If a user requires root-level access, there's always su(1). Or sudo, or pfexec, or any number of privilege management tools. SSH-based orchestration systems like Ansible support all of these programs. Sudo in particular can be configured to authenticate via an SSH agent, so that the users' credentials are never exposed to the server.

Certain environments, particularly large cloud-based server farms, are designed so that logging in as `root` is not only possible but preferable. These environments require public key authentication and log the key used to authenticate each session. Most readers of this book do not work in that environment. We'll look at setting that up in Chapter 14, "Certificate Authorities."

OpenSSH controls direct login as root with the PermitRootLogin keyword. By default, `sshd` permits direct root logins if they're done with public key authentication.

```
PermitRootLogin prohibit-password
```

The `prohibit-password` option is the same as the older but confusingly-named `without-password`. Users can log in as root, so long as they don't use a password to do it. Once you get into public key authentication, nothing prohibits a user from adding their key to the list of keys permitted to use the root account. I advise against using `prohibit-password`.

Setting PermitRootLogin to `no` disallows direct logins by root. Most operating systems set this by default.

If you must allow remote root logins, consider setting PermitRootLogin to `forced-commands-only`. Chapter 12 discusses the ForceCommand option, letting you restrict automated tasks that must run as root to only perform certain commands.

Logging in as root via SSH almost always means you're solving the wrong problem. Step back and look for other ways to accomplish your real goal.

Tokens

Certain keywords in *`sshd_config`* can also use tokens, symbols that represent some variable. Tokens make these keywords much more flexible. We'll talk about using tokens when we discuss the keywords that can use them, but from the start you need to recognize them on sight. We'll use tokens when building chroots in the next section, and then throughout this book.

All tokens start with a percent sign (`%`). The simplest token is `%%`, which stands for an actual percent sign. If you have file paths with a percent sign in them, you might need this.

The token `%u` represents the username.

The token `%h` represents the user's home directory.

Most of the other tokens are used only in very special circumstances, when using less common functions. We'll touch on them as needed, but these are the ones everyone must know. The sshd_config(5) man page lists all the tokens.

Chrooting Users

At times a user needs access to a command prompt or a specific program, but

you don’t want the user to access anything outside his home directory. A directory the user cannot escape is called a *chroot*. (A chroot is also useful for SFTP, as discussed in Chapter 6, but that requires much less configuration.) OpenSSH supports chrooting users with the ChrootDirectory option.

```
ChrootDirectory none
```

By default, `sshd` does not chroot users.

Populating a Chroot

A chrooted user cannot access anything outside the chroot. Any chroot you create will not have device nodes, shells, or other programs unless you place them there. When your restricted user logs in, `sshd` will fail to find a shell or home directory and immediately disconnect them. To give a chrooted user shell access you must at minimum set permissions on the chroot directory, create a home directory for the imprisoned user, create device nodes, and install a shell.

You only need to populate the chroot if the user needs shell access. If the user only gets file copy access via SFTP, the ForceCommand keyword discussed in Chapter 6 is preferable to a populated chroot.

The chroot directory must be owned by **`root`** and not writable by the restricted user, just as you would not permit an unprivileged user to write to the host’s root directory. If the restricted user can write to the chroot directory, `sshd` will not let them log in.

A user’s home directory (as shown inside */etc/passwd*) is expected to be available inside the chroot. If user **`pkdick`**’s home directory is listed as */home/pkdick*, and he is chrooted into */usr/prisonroot*, you must create the directory */usr/prisonroot/home/pkdick*. This directory should be owned by the user, just like a regular home directory, and should contain any necessary dotfiles.

Create a device node directory inside the chroot. With a chroot directory of */usr/prisonroot*, you’d need */usr/prisonroot/dev*. Now you need to populate this with device nodes. A chroot doesn’t require a full complement of device nodes, but most chrooted applications need at least */dev/random*, */dev/stdin*, `/dev/stdout`, */dev/stderr*, */dev/tty*, and */dev/zero*. The method to create device nodes varies between operating systems. OpenBSD and many Linuxes use a shell script `/dev/MAKEDEV`, while FreeBSD and many commercial Unix-like systems use a device filesystem. Check your operating system to see what device nodes a chroot needs and how to create them. Some operating systems include tools to easily populate a chroot.

Finally, users need a shell. Copy a statically-linked shell into the chroot’s */bin* directory. Also copy static versions of any other programs the user needs. If you want to use dynamically linked programs, you must also copy over any necessary files.

Assigning Chroot Directories

Use the ChrootDirectory option to establish chroots.

```
ChrootDirectory /home/djm
```

This works for a single user account, or if all SSH users have the same chroot directory, but this is a place where tokens come in useful.

If your chroot directory path includes a literal percent sign, use the `%%` token. Here we chroot into the directory */home/disk%1/djm*.

```
ChrootDirectory /home/disk%%1/djm The %h macro expands to the user's home directory, as
specified in /etc/passwd.
ChrootDirectory %h
```

At login, djm gets locked into */home/djm*. Note that he'll need a chrooted home directory inside this directory, so you'll need to create */home/djm/home/djm*.

The `%u` macro expands to the user's username. This lets you assign a group of users unique home directories under central chroot directory.

```
ChrootDirectory /usr/prisonroot/%u You'll need to populate each user's chroot separately.
```

Choosing Users to Chroot

You can chroot everyone, but that would make it hard for your sysadmins to perform maintenance. Chances are you only want to chroot a subset of your users. Use a Match statement to selectively chroot users.

```
…
ChrootDirectory none
…
Match Group billing
ChrootDirectory %h
```

If a majority of your users are chrooted, reverse the default to allow only your sysadmins full access.

```
…
ChrootDirectory %h
…
Match Group wheel
ChrootDirectory none
```

Choose whichever method makes sense for your environment.

Debugging a Chroot

Chroots are difficult to manage in that they normally lack a complete userland. If a chrooted user cannot log in, run `sshd` in debugging mode, attached to a terminal window. Have the chrooted user attempt to log in, and watch the debugging output; you'll probably see the problem. Common issues include missing device nodes, incorrect directory permissions, or a missing shell.

Protecting the SSH Server

Any Internet-facing server will have lots of random stuff poking at it. Worms, script kiddies, and other assorted scum would really like to break into your computer. If nothing else, someone wants to run an IRC bot on it. How can you protect your SSH service?

Some people recommend changing the TCP port that `sshd` uses. This is a

perfect example of security through obscurity, which does not work. Scanners constantly probe all ports of all Internet-connected IP addresses, and they're pretty good at figuring out what service is running on which port. Changing ports might buy you a couple of minutes against a dedicated intruder, but no longer. Changing ports *can* reduce the amount of random noise you get in your logs, increasing the odds of you noticing real problems.

You'll also see random folks on the Internet recommend using a different protocol banner, which is a poor idea. You'll see the protocol banner when you use netcat to connect to the SSH daemon. The banner identifies the type of server. All SSH servers differ slightly, and might require special client settings. SSH clients use the protocol banner to detect any quirks needed for a reliable connection with a server. If you change the protocol banner from SSH-2.0-OpenSSH_7.0 to SSH-2.0-ParanoidWhackJob, you're depriving clients of information they need to reliably connect.

You might also consider add-on solutions to block IP addresses that repeatedly connect but fail to authenticate, such as fail2ban and blacklistd. The details of implementing these varies widely by platform, so I'm not going into them, but they are worth considering.

To some extent, sshd(8) protects itself via *privilege separation*. Only a small section of the service runs with `root` privileges. Most of the server runs as an unprivileged user. This means that if an intruder successfully breaks into the server daemon, he can only do a limited amount of damage to your system. It's still really annoying, but not devastating.

Additionally, sshd(8) restricts the unprivileged process via a *sandbox*. The sandbox restricts which syscalls `sshd` can call before the user authenticates. OpenSSH supports a few different sandbox methods, from Apple's sandbox(7) to Linux's seccomp(2). If the operating system doesn't offer any other sandboxing methods, `sshd` uses rlimit to set the number of open files and child processes to zero.

As with all Internet-facing services, a simple way to reduce risk to your SSH service is to reduce the number of IP addresses that can access it. OpenSSH respects TCP wrappers (`/etc/hosts.allow`). If your server or network has a packet filter, use it instead. By only allowing authorized IP addresses to access your SSH server, you block the vast majority of attackers.

The most effective way to protect your server, however, is to disable passwords and only allow logins via keys. We cover access via keys in Chapter 7, "SSH Keys."

We'll return to configuring `sshd` when we cover specific features, but for now let's talk about server keys.

[1] And you need to inflict bodily harm until privilege separation gets turned back on.

[2] If you don’t like your vendor’s answer, ask more loudly and with malice aforethought.

[3] The impact on your employment of evading the corporate firewall is left as an exercise for the reader.

Chapter 4: Verifying Server Keys

If you're paranoid, or if you've been a sysadmin for longer than a week, you need to be sure that the server you're logging into is the server you think you're logging into. Server keys help verify a server's identity before you exchange authentication information with the wrong machine.

Network connections over unencrypted, unauthenticated protocols are easily diverted to the wrong machine. An intruder who controls a publicly accessible device, such as a server, can make it spoof a different server's identity. Every user that logs onto the spoof server gives his username and password to the intruder. Often the intruder will then forward the session to the actual destination host, so that the user never realizes that they've been caught. This is a classic network attack that is still widespread today; the protocols change, the applications change, but man-in-the-middle attacks and spoofing are forever.

When properly deployed and used, SSH categorically eliminates these attacks. Even if an intruder can make one machine resemble another, even if he copies the login prompts and the web site and the operating system version, the intruder cannot copy the target server's private key unless he already controls the server. Without the private key, the spoof server cannot decrypt anything transmitted via the server's public key.

SSH server keys verify the server's identity to the client. They are *important*, not something you just hit ENTER to accept.

Every SSH server has one or more unique public keys, as discussed in Chapter 1. The first time an SSH client connects to an SSH server, it displays the server's public key fingerprint to the user. The user is expected to compare the fingerprint shown with the server's key fingerprint. If they match, the user tells their SSH client to cache the key and the connection continues. If the keys don't match, the user terminates the connection.

On all subsequent connections to the server, the client compares its cached key to the key presented by the server. If the keys match, the connection continues. If the keys don't match, the client assumes that something has gone wrong and requests user intervention.

For SSH server keys to be useful, you must verify that the key displayed by the client is identical to the key offered by your target server. A public key is several hundred characters long, however. Sysadmins can't realistically ask users to compare hundreds of characters to a list of known-good keys; most users automatically dismiss the task as impossible. Explaining that it's very possible, but very tedious and very annoying, does not improve the discussion.

SSH summarizes public keys with key fingerprints.

Key Fingerprints

A key fingerprint is an almost human-readable summary of a public key. Any user can get the public key fingerprints; if you need the private key fingerprints, you'll need to be **root**. View a key's fingerprint with the ssh-keygen(1) program, using `-l` to print the fingerprint and `-f` to specify a key file. Here I view the fingerprint of this host's ED25519 key.

```
$ cd /etc/ssh
$ ssh-keygen -lf ssh_host_ed25519_key.pub
256 SHA256:JwmD+yFwH83rPdhorge/S6qxXAUy3/G0CvFqTrcIWkY root@www (ED25519) We see that this
key use 256-bit SHA-256. The fingerprint itself is the long string beginning with JwmD…
and ending with cIWkY. After that we have the user and host that generated the key, plus
the key type in parenthesis.
```

The server and client negotiate on which key to use for a connection. The client might present any supported key to the user, so you'll need the fingerprint of every public key on the server. The easiest way to collect all the fingerprints is to copy them to a file.

```
$ ssh-keygen -lf ssh_host_ed25519_key.pub > $HOME/fingerprints.txt
$ ssh-keygen -lf ssh_host_ecdsa_key.pub >> $HOME/fingerprints.txt
$ ssh-keygen -lf ssh_host_rsa_key.pub >> $HOME/fingerprints.txt
```

Now get those fingerprints to your users.

You can use ssh-keyscan(1) to retrieve key fingerprints from your SSH servers, but you must verify those fingerprints against the server's public key. By the time you do that, you might as well extract the public key fingerprint from the server itself. The ssh-keyscan program is useful for verifying that a host's public key fingerprints haven't changed, however.

Making Host Key Fingerprints Available

A user first connecting to an SSH server should compare the host key fingerprint that appears in their client to a known good host key fingerprint. Real users only do this if the comparison process is easy, though. The system administrator needs to make fingerprint comparisons simultaneously easy and secure. The easiest way is probably to display the key fingerprints on an encrypted Web site accessible from within your organization. When an employee needs SSH access to the server, give them a link to the fingerprint page when you give them their login credentials. Do not distribute key fingerprints over insecure media, such as email or an unencrypted Web site.

Chapter 11 offers methods to automatically distribute keys and fingerprints. Deploying these methods eliminates the need for users to manually verify keys, simultaneously increasing compliance and decreasing everyone's workload.

If you're running the OpenSSH client, you can simplify key verification with key certificates (Chapter 14), SSHFP records (Chapter 11), or both. Very few other clients, including PuTTY, support these protocols.

Host Keys and the OpenSSH Client

When you first connect to an SSH server with the OpenSSH client ssh(1), you'll see a prompt requesting that you verify the key.

```
$ ssh gluttony
The authenticity of host 'gluttony (203.0.113.213)' can't be established.
ECDSA key fingerprint is SHA256:jovou1bQ0S1Ex6QBjo4T+0+FzwzyTXLqxF/aPudVTnk.
No matching host key fingerprint found in DNS.
```

This is your opportunity to verify that the OpenSSH server is actually the host you think it is. OpenSSH offers you the ECDSA key fingerprint. Grab your list of server keys and compare the ECDSA key fingerprint in the list to the ECDSA key fingerprint in the client. If the key fingerprints match, type `yes` to cache the verified key and continue the connection. You'll get a message much like the following.

```
Warning: Permanently added 'gluttony' (ECDSA) to the list of known hosts.
```

The next time you connect to this host, ssh(1) will compare the cached host key to the host key on the server and either silently and securely connect, or loudly and securely disconnect.

If the key does not match, ssh(1) immediately disconnects without caching the key. Immediately notify your sysadmin and/or security team that the host key does not match.

OpenSSH also supports an easier way to compare key fingerprints, called *randomart*. A randomart image is a visual interpretation of a key fingerprint. It's a non-standard representation, however. Feel free to experiment with randomart, but don't assume it's universally available.

Host Keys and the PuTTY Client

The first time you connect to a server with PuTTY, you'll get a warning much like Figure 4-1.

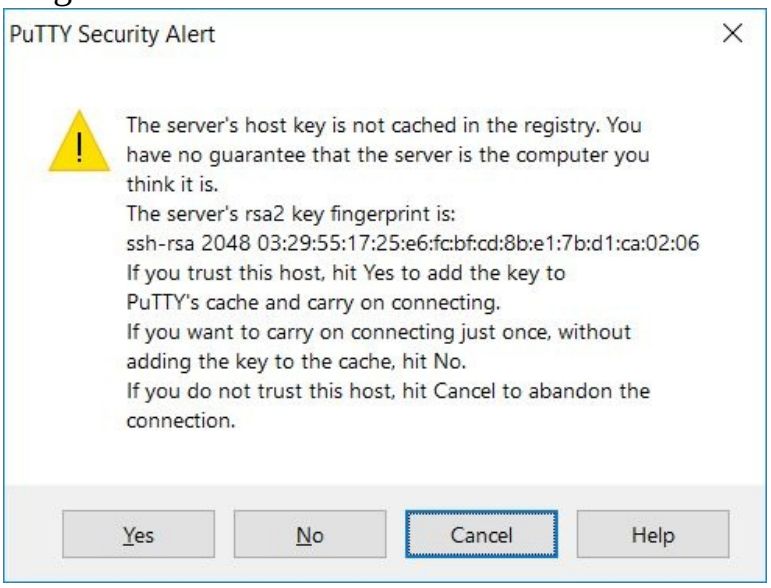


Figure 4-1: PuTTY Key Fingerprint Message Compare the key fingerprint shown in the client to the key fingerprint in your list. Note that PuTTY negotiated a connection using an RSA key, which is different that the ECDSA key agreed on between OpenSSH and its OpenSSH server.

If the keys match and you want PuTTY to cache the key for future reference and then connect, hit Yes.

If the keys match, and you want PuTTY to connect without caching the key, hit No.

If the keys do not match, hit Cancel to terminate the connection. The host you're connecting to is not the host you think you're connecting to. Verify that you entered the correct hostname, then notify your sysadmin and/or security team of the non-matching host key.

When Keys Don't Match

If a host key has changed, you'll get a message much like this.

```
$ ssh gluttony
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@ 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 a host key has just been changed.
The fingerprint for the ECDSA key sent by the remote host is
SHA256:TSJ39GppnUdf8JX6J0oAf9+Cga2LzLNXX+tid54lfo4.
Please contact your system administrator.
Add correct host key in /home/mwlucas/.ssh/known_hosts to get rid of this message.
Offending ECDSA key in /home/mwlucas/.ssh/known_hosts:5
ECDSA host key for gluttony.mwl.io has changed and you have requested strict checking.
Host key verification failed.
```

Scary-looking stuff? It should be. Something scary has happened. Your SSH client is screaming that something is Very Wrong. If your laptop was an ambulance, the lights would be spinning and the siren blaring. The super-secret host key pair used to identify this system has changed. This can happen for one of six reasons. Identifying which requires talking to the sysadmin.

Maybe the sysadmin destroyed the key pair, either accidentally or deliberately, and generated a new key pair. She should have a new key fingerprint for you.

Perhaps the key fingerprint cached by your client is wrong. You might have a desktop security issue.

Or, the server might have been upgraded or replaced, and now supports a new key algorithm. The sysadmin should have a new key fingerprint for you.

It could be that the site uses round-robin DNS, effectively giving several servers a single hostname, and you're connecting to the shared name rather than an individual server's unique name. Access individual hosts, not a shared hostname.

Perhaps your host's key cache is corrupt. Re-validate the host key. This is annoying, but not insurmountable. Your sysadmin can confirm that the host key

has not changed.

Lastly, it's possible that an intruder controls the server or has diverted your connection to a different server. You, the sysadmin, and/or your security team, are about to have a bad day.

The only way to know which? Talk to the sysadmin. DO NOT CONNECT TO THE SERVER UNTIL YOU KNOW WHY THE KEY CHANGED. All of these are serious errors that require investigation.

If the key changed for a legitimate reason, verify the new key. If the new key is correct, replace the old key with the new one. PuTTY offers to replace the key for you, while in OpenSSH you must edit the key cache yourself, as discussed in Chapter 5. The error message gives the line in `known_hosts` that contains the obsolete key. If the key is still not correct, talk to the sysadmin again. A legitimate SSH key change might mask an illegitimate intruder; I've seen more than one freshly installed server get compromised before the first legitimate logon.

You can override the SSH client's refusal to connect to machines when the host key changes, or not cache the new key, but remember, SSH doesn't just validate the server and protect your data in transit. A completed connection also hands your authentication information to the SSH server. If you give your username and password to a compromised machine, you've just given the intruder your username and password. If you use the same password on multiple machines, you can no longer trust any of them. Cancel your weekend plans right now, and possibly next weekend's as well. You'll be busy recovering from backup and managing irate customers.

A mismatched key message is a sign that SSH works. Use it.

Chapter 5: SSH Clients

SSH client software resides on a user's workstation and permits connections to an SSH server. We'll discuss two common clients: the OpenSSH command-line client for Unix-like hosts, ssh(1), and the PuTTY client for Microsoft Windows. Both clients are freely usable and redistributable, in source or binary form, with very minimal restrictions or limitations.

People have written other SSH clients, of course. You can get an OpenSSH-based client for Windows systems either through Cygwin or Microsoft's Windows Subsystem for Linux on Windows 10 and newer. There's a straight port of OpenSSH to Windows (https://github.com/PowerShell/Win32-OpenSSH). Microsoft has released a beta port of OpenSSH to Windows 10 and newer, as a developer feature. Similarly, PuTTY has been ported to many Unix-like systems and mobile devices. Many people have forked both PuTTY and OpenSSH, modifying them to fit their needs. Many of these are solid, reliable projects. Once you have a solid grounding in SSH, feel free to use the client that you prefer.

Each client has its own section in this chapter. Further chapters involving SSH clients will get chopped into three sections: one for the theory of what we're doing, followed by separate sections on configuring each client.

OpenSSH Client

The OpenSSH client, `ssh`, is developed synchronously with the OpenSSH server. As new features often appear in OpenSSH before other SSH implementations, you'll get the bleeding edge of SSH features by using the newest OpenSSH client. The OpenSSH client is developed as part of OpenBSD, but a new portable release appears every six months.

A user's personal SSH settings are recorded as files in *`$HOME/.ssh/`*. Like the home directory, this directory must be writable only by the user and **`root`**, although you can allow it to be world-readable. Various client and server functions stop working if others can write to this directory. While `ssh` creates *`$HOME/.ssh`* with correct permissions, if your SSH suite behaves oddly check the permissions.

To run `ssh`, enter the command follow by the host you want to connect to.

```
$ ssh gluttony.mwl.io
```

This uses your client's default settings to connect to the host **`gluttony.mwl.io`**, including your current username.[1]

If `ssh` doesn't behave as you expect, try running it in verbose mode with `-v`. You'll see the server and client negotiate protocol version and encryption

algorithms, the server present its host key, the client verify that key, and the two negotiate authentication methods. While this might not solve your problem, it will tell you where the login fails and give you a hint about where to look. Reading the output carefully might tell you that, for example, the server only permits logins with public keys or you're trying to use an unsupported encryption method.

```
$ ssh -v gluttony.mwl.io
```

If you still have trouble, multiple `-v` options increase the debugging level.

In normal cases, that's it. The rest of this book is about abnormal cases.

OpenSSH Client Configuration

Configure `ssh` by setting options, either on the command line or in a configuration file. Use configuration files for permanent changes and the command line for temporary ones. We'll look at the configuration file first.

Two files control ssh(1) behavior: */etc/ssh/ssh_config* and *$HOME/.ssh/config*. Each contains keywords and values, as discussed in Chapter 2. The former establishes default behavior for all system users. The latter is the user's personal SSH client configuration. A user's configuration overrides all global settings, but most users can't be bothered to enter their own custom configurations. Configuration file changes affect all SSH sessions started after the change. There's no process to restart, but changing the configuration doesn't affect existing SSH sessions. Both files have the same syntax and accept exactly the same options. I'll refer to *ssh_config* for brevity, but everything applies equally well to *$HOME/.ssh/config*.

While most connection options can be set on the command line, I recommend storing permanent information in *ssh_config*. Programs such as scp(1) and sftp(1) (see Chapter 6) read *ssh_config*, and each of these programs have slightly different command line options. Using a configuration file centralizes configuration.

The user's personal configuration overrides the global configuration. Options set on the command line override both.

Per-Host Configuration

You can use the Host keyword to change how `ssh` connects to certain hosts. Here, I use the Port keyword to change the TCP port `ssh` connects to, but only for hosts in the `mwl.io` domain. It uses port 22 for all other hosts, as specified in */etc/services*.

```
Host *.mwl.io
Port 2222
```

I could also specify an IP address, or a network of IP addresses.

```
Host 192.0.2.*
Port 2224
```

Note that `ssh` matches these *ssh_config* entries based on what the user enters on the command line. Host entries must be an exact case-sensitive match for what

the user types. Assume that my *ssh_config* contains both Host entries above, and let's see how this works in practice.

```
$ ssh gluttony.mwl.io
```

This matches the first Host entry, so ssh connects to port 2222.

My desktop's */etc/resolv.conf* automatically appends the domain **mwl.io** to any lone hostnames, so I probably wouldn't type the fully qualified domain name. Instead, I'd just do something like this.

```
$ ssh gluttony
```

This won't match my first Host entry, as I didn't explicitly type the domain name given in *ssh_config*. If the host gluttony has an IP address in 192.0.2.0/24, though, wouldn't the second Host entry match? No, because the Host entries match on the command line; there is no check against DNS. To match based on the IP address in the Host entry, I would need to explicitly run `ssh 192.0.2.whatever`. Custom settings for this host require a Host entry like this.

```
Host gluttony
Port 2222
```

Conditions are parsed on a first-match basis. Configuration options listed after Host entries remain in effect until the next Host entry. This *ssh_config* is probably wrong.

```
Host *.mwl.io
Host 192.0.2.*
Port 2222
```

The user probably wanted the Port keyword to apply to all hosts in **mwl.io** and all IP addresses in 192.0.2.0/24. We have an entry for any host in the **mwl.io** domain, but there's no special configuration for it. Any hosts in 192.0.2.0/24 run sshd on port 2222. Instead of doing this, list multiple hosts on the same line, separated by spaces. Here I list my domain name, my IP addresses, and my servers.

```
Host 129.0.2.* mwl.io *.mwl.io gluttony avarice lust pride wrath envy sloth Port 2222
```

I list both *.mwl.io and mwl.io because there is a specific machine named **mwl.io**. The leading asterisk and period before the domain name will not match that host.

Put any global defaults at the beginning of your configuration file. Suppose your organization has a policy of running SSH on port 981, because they like security through obscurity, but your special servers use a different port for even more obscurity.

```
Port 981
Host *.mwl.io
Port 2222
```

Here the default port is 981, but the specified hosts use port 2222.

Sometimes you want to test changes without mucking with a working configuration, or maybe you have an automated process that needs a special configuration file. To use a configuration file other than *ssh_config*, specify it on

the command line with the `-F` option.

```
$ ssh -F test-config avarice
```

You can now experiment with features without breaking your working configuration.

If you have enough hosts, you might consider establishing canonical hostnames in *`ssh_config`*.

Canonical Hostnames

On a large enough network, or in an orchestrated environment where servers are dynamically created and destroyed, listing all of your SSH servers quickly becomes unrealistic. The CanonicalizeHostname keyword tells `ssh` to rewrite standalone Host entries in *`ssh_config`* into specific domains, and then (if they exist) use that hostname for configuration and key management. This lets you eliminate many lengthy Host keywords. Set CanonicalizeHostname to `yes` and CanonicalDomains to your domain. Consider the following configuration: CanonicalizeHostname yes

```
CanonicalDomains mwl.io
Host *.mwl.io
Port 2222
```

The next time I run `ssh gluttony`, `ssh` checks to see if there's a hostname **`gluttony.mwl.io`**. If that hostname exists, `ssh` evaluates *`ssh_config`* as if I'd run `ssh gluttony.mwl.io`. This connection gets the special rules that apply to hosts in the mwl.io domain.

You can list multiple canonical domains. The canonical names are tested in the order they're listed, and the first match wins. Consider an entry like this.

```
CanonicalDomains mwl.io michaelwlucas.com When I run ssh wrath, ssh(1) searches for
wrath.mwl.io. If it finds that host, it opens a connection. If it can't find that host,
ssh searches for wrath.michaelwlucas.com.
```

If you activate hostname canonicalization, `ssh` defaults to trying to canonicalize hosts with one or fewer dots in them. This lets canonicalization catch subdomains, like **`www.detroit.mwl.io`** for **`www.detroit`**. To change the maximum number of dots in the hostname, use the CanonicalizeMaxDots keyword. Here I allow zero or fewer dots.

```
CanonicalizeMaxDots 0
```

OpenSSH has a few other hostname canonicalization features, discussed in ssh_config(5).

Common SSH Options

The most common features people use are changing the username, the port, or adding SSH options.

Changing Usernames

Most SSH clients assume that your username is identical on both the client and server, and tries to log into the remote system with the same username you have

on the local machine. On most of my systems, my username is `mwl`. Occasionally someone creates an account for me with a different username, like `mlucas` or `lucas` or `michael` or `jerkface`. I must tell `ssh` to use that username on the remote system. Do this by putting the user account name, followed by the @ symbol, then the remote machine name.

```
$ ssh jerkface@devio.us
```

You can also specify a username with `-l`.

```
$ ssh -l jerkface devio.us
```

If this is an ongoing thing, specify the username in *`ssh_config`* with the User option.

```
Host devio.us
User jerkface
```

By storing usernames in *`ssh_config`*, I can forget about them and free up valuable brain space.

Changing Port

Some sites run SSH on a port other than 22, usually to provide an appearance of improved security. It doesn't actually secure SSH, but it does reduce log noise. Use `-p` and a port number to change the port `ssh` connects to. If your server runs `sshd` on port 2222, connect with: $ **ssh -p 2222 gluttony**

You can specify the port in *`ssh_config`*.

```
Port 2222
```

Again, I recommend storing permanent connection information in *`ssh_config`*.

SSH Options on the Command Line

SSH isn't just a command; it's a protocol. And that protocol has all sorts of edge cases. Sometimes you'll need to set some of those edges on the command line. While everything OpenSSH supports is available as an *`ssh_config`* keyword, not all of those keywords have command-line equivalents. To set those keywords on the command line use the `-o` command-line option, the option name, an equals sign, and the value of that keyword.

```
$ ssh -o Port=2222 sloth
```

This example is trivial—the Port keyword has a dedicated command-line option, `-p`. We'll see more complicated examples later.

Evaluating your SSH Configuration

You can set command-line options, options in the user's configuration file, and options in the global client configuration file. Host keywords can muck up your carefully adjusted defaults, or your carefully adjusted defaults can require you to use Host keywords for specific servers. How do you know what options ssh(1) is really using when you connect to a host?

Use the `-G` option to `ssh`. It tells ssh(1) to parse all the configurations for the target host, print out the configuration it's going to use, and immediately exit without connecting. You can review your settings to verify you're getting what

you need.

SSH Jump Hosts

Sysadmins often have to pass through one host to get to another. Maybe you trust this intermediate host. Maybe you don't. OpenSSH supports *jump hosts*, letting you use an SSH server as a relay to connect to a second server. Yes, you could do this manually by logging into the intermediate host and running `ssh` again, but using the built-in support means that the jump host sees none of your plain text. The jump host has no control over the options your client and the target server negotiate. This means you can, say, forward X or your SSH agent through a jump host that accepts neither.

Specify a jump host with `-J`. Add the username if needed.

```
$ ssh -J mwl@envy jerkface@pride
```

I'm trying to log in as **`jerkface`** on the host **`pride`**, using the account **`mwl`** on host **`envy`** as a jump host. I'll get prompted for my authentication credentials on the jump host, and then my credentials on the destination.It's best to use public key authentication on both hosts.

Set a jump host in *`ssh_config`* with the ProxyJump keyword.

```
Host pride.mwl.io
ProxyJump mwl@envy.mwl.io
```

How much do you have to trust your jump host? None of your keystrokes reach the jump host, so you don't have to worry about session logging. The only thing the jump host can see is an encrypted data stream between your client and the destination server. The jump host could alter or interrupt the encrypted stream, but that's exactly the sort of tampering SSH is designed to detect.

Some Linux distributions disable jump hosts in their client.

Addressing Options

The OpenSSH client lets you choose how it uses TCP/IP, by setting the address family and the source address.

IP Protocol Version

Hosts can have both IPv4 and IPv6 addresses. The AddressFamily keyword tells the client to connect with only IPv4 (inet) or with only IPv6 (inet6). The default is `any`, which means "connect over whichever protocol the system resolver returns an address for." Sometimes, you'll have better connectivity over one protocol or the other. If you get your IPv6 connectivity via a tunnel, using only IPv4 for SSH might make sense. Similarly, if you have unlimited IPv6 connectivity, you might want to use IPv6 for everything. Here I deliberately disable IPv4.

```
AddressFamily inet6
```

You can choose to use only IPv4 with the `-4` command-line option.

```
$ ssh -4 lust
```

Force IPv6 with `-6`.

Set Source Address

Hosts with multiple IP addresses on a single interface default to originating all connections from that interface's primary IP address. This is not always desirable. Services can migrate from host to host, often independently of any firewall changes. You can tell `ssh` to use a source IP address other than the primary with the BindAddress keyword in *`ssh_config`*.

```
BindAddress 192.0.2.91
```

The BindAddress must be attached to the local machine.

BindAddress has no convenient command-line flag. You must specify it with `-o`.

The OpenSSH Host Key Cache

The OpenSSH client records host keys approved by the user in *`$HOME/.ssh/known_hosts`*. Each key appears on its own line in *`known_hosts`*, much like this.

```
wrath.mwl.io ecdsa-sha2-nistp256 AAAAE2VjZHNhLXNoYTItbm…
```

Each line contains the machine's hostname (**`wrath.mwl.io`**), host key type (ecdsa-sha2-nistp256), and the public key itself.

Key Caching

How do you want to update your key cache? In some environments, users must manually verify host keys and then manually add them to the key cache. In other environments, it's acceptable to automatically add new keys to the cache. Most commonly, users want `ssh` to ask them what it should do. The StrictHostKeyChecking *`ssh_config`* option tells `ssh` how to treat new host keys.

If you want `ssh` to refuse to connect to any host that doesn't have an entry in *`known_hosts`*, set StrictHostKeyChecking to `yes`. The only way for the client to connect is to add the host key to the *`known_hosts`*, presumably from a central repository provided by the sysadmin. This makes most sense in an environment where host keys are automatically distributed.

If you're at the opposite extreme, and you will never verify a host key no matter how important it is, you might as well set StrictHostKeyChecking to `accept-new`. This tells `ssh` to blindly update *`known_hosts`* with every new key it gets. This is the equivalent of never bothering to lock your home, car, office, and bank vault—it might feel airy and freeing, but sooner or later someone's going to take uncivil liberties with your personal belongings.

The default setting, `ask`, tells `ssh` to present any unknown keys and ask the user what to do. You can verify the key, accept it, and have `ssh` add it to *`known_hosts`*, or reject the host key.

Choose the option that best suits your environment. Your laptop probably has

different needs than a server run by the NSA or a criminal cartel, and all of those are different from the orchestration system in your test lab.

Cache Security: Hashing known_hosts

The *known_hosts* file comes in really handy to intruders who break into your desktop; it's a convenient list of SSH servers to target. As your SSH servers might share a common sysadmin, the technique used to penetrate your desktop might work on any of those servers. Additionally, sysadmins and other users can view the contents of *known_hosts*. The best way to prevent snooping is to change *known_hosts* so that it no longer contains a list of hostnames. Accomplish this by hashing the hostnames, exactly as */etc/passwd* does with passwords.

If you replace the hostnames with hashes, nobody can read the host names from the file, nor can anyone reverse-compute the hostnames. When you connect to a host, however, `ssh` can easily compute the hash of the server hostname and look up that hash in *known_hosts*.

A hashed *known_hosts* entry looks something like this.

```
|1|PBM07JCRBjfg8qOz1BokTtCDly0=|DVXu0IFq/dC4GMfbEbfVkhptVjQ= ecdsa-sha2-nistp256
AAAAE2VjZ…
```

If you examine the entry, you'll see the key algorithm and the host key fingerprint further down.

To have `ssh` automatically hash new host keys added to *known_hosts*, use the *ssh_config* keyword HashKnownHosts.

```
HashKnownHosts yes
```

This will not hash existing entries, however. Use the `-H` flag to ssh-keygen(1) to hash your existing *known_hosts*.

```
$ ssh-keygen -H
/home/mwlucas/.ssh/known_hosts updated.
Original contents retained as /home/mwlucas/.ssh/known_hosts.old WARNING:
/home/mwlucas/.ssh/known_hosts.old contains unhashed entries Delete this file to ensure
privacy of hostnames Hashing your known_hosts copies the existing cache to
known_hosts.old, then hashes everything inside known_hosts. Verify that ssh can still
connect to all your usual hosts. Once you feel confident that your key cache is still
usable, delete the unhashed known_hosts.old.
```

To find a single host entry in the hashed *known_hosts* file, use `ssh-keygen -F` and the target hostname.

```
$ ssh-keygen -F avarice.mwl.io
# Host avarice.mwl.io found: line 17
|1|5hcRwDHWxwxCWrFTngG4jT4OhJ0=|TyJXB6z+oEJXSP5MzakulFWgPDI= ecdsa-sha2-nistp256…
```

You now know that this entry is on line 17 of the file, and can easily copy it.

To remove a hashed hostname, use `ssh-keygen -R`.

```
$ ssh-keygen -R avarice.mwl.io
# Host avarice.mwl.io found: line 17
/home/mwlucas/.ssh/known_hosts updated.
Original contents retained as /home/mwlucas/.ssh/known_hosts.old If you hadn't deleted the
unhashed known_hosts.old, well, it's gone now.
```

When distributing *known_hosts* from a central system (Chapter 11), there's no reason not to provide the hashed version.

The PuTTY Client

PuTTY is an SSH, telnet, and serial client, as well as a terminal emulator, for both Windows and Unix-like systems. It's available at https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html, or a Web search for "putty SSH" will take you right there. While it's not written by the professional paranoiacs in the OpenBSD team, PuTTY's freely-available source code has been repeatedly audited. PuTTY is probably the most widely deployed Windows SSH software.

The PuTTY download page offers several choices. I recommend the full installer that contains PuTTY and all its related programs. You won't need everything, but it'll be easier and faster than downloading various utilities individually. PuTTY doesn't truly need an installer; you can download plain PuTTY.exe if you prefer. The installer does create shortcuts in the Start Menu, registers the programs with the operating system, and handles all the other Windows minutiae.

PuTTY can run on the command line. The arguments and command-line flags are conveniently similar to those of OpenSSH. If you're running Windows, though, you're probably interested in PuTTY as a graphical program. We'll focus on the pointy-clicky interface, but you should know that the command line is an option if needed.

If you feel adventurous, you could download the PuTTY development snapshot instead. This includes all of PuTTY's latest patches and features, but it might also contain brand-new bugs.

Start PuTTY and you'll get a screen like Figure 5-1.

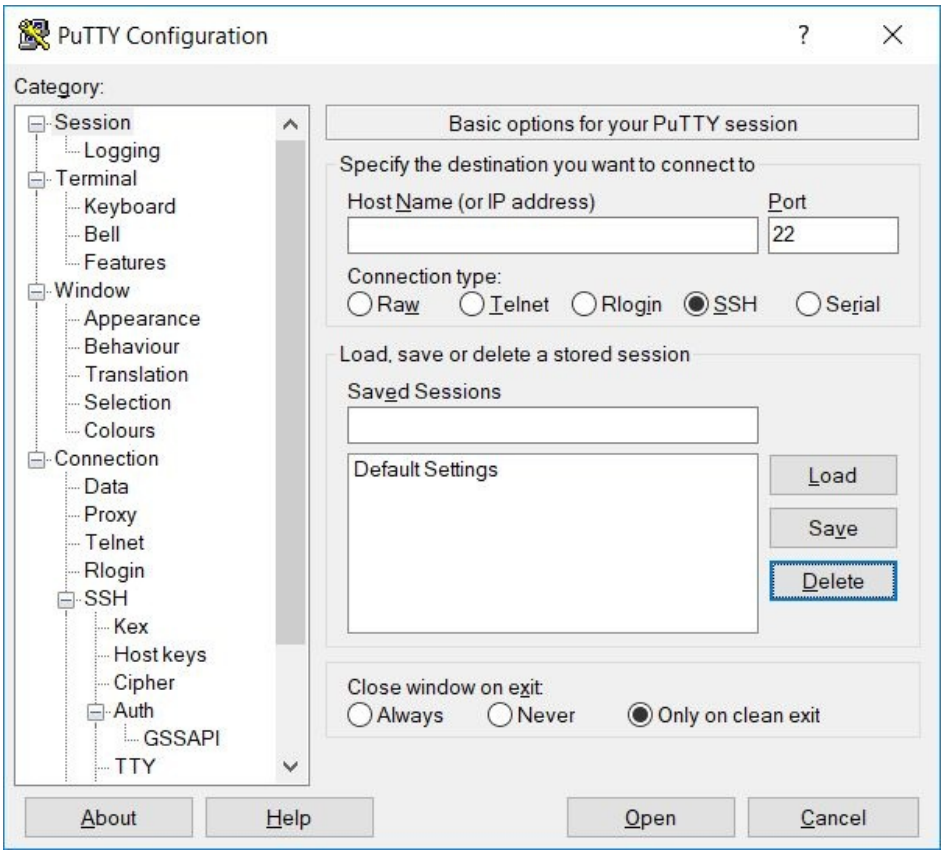


Figure 5-1: PuTTY Startup Screen On the right side you'll configure connections to different servers. We only have one connection right now, *Default Settings*. It won't connect to anything, but you'll use it to set your PuTTY defaults. You can always get back to this screen by hitting *Session* on the left-hand side.

On the left side, you configure options on how PuTTY presents itself and how it handles supported protocols. PuTTY supports a variety of protocols. If you need a flexible general-purpose terminal emulator, PuTTY can probably meet your needs. We're only going to cover SSH, however. Note the *SSH* option, second from the bottom. Click there, or expand that little "plus" sign, to view and edit details on how PuTTY performs SSH.

If you select something on the left side, the right-hand side changes to show details on your selected option. Select *SSH*, and the right-hand side shows a few basic protocol options, such as the protocol version and sharing SSH connections (see "Connection Multiplexing" later this chapter.) We'll use these settings to establish our PuTTY defaults.

Setting PuTTY Defaults

Your fresh PuTTY install lists only one connection, Default Settings. Every new connection you create starts by copying everything in *Default Settings*. Click on the *Default Settings* connection and hit *Load*. You can then edit and save the *Default Settings* connection.

Start by setting a default username. My accounts on my work system all have

the same username. The only time my username varies is when it's an account on an external system. I can set my default username in PuTTY, and have it pre-configured to my most common setting. In the left side, choose *Connection*. Under *auto-login username*, I put my standard username.

Now go back to the *Session* panel. Select *Default Settings*, and hit *Save*. You've updated your default connection. Repeat the process to make further changes to your defaults.

Starting SSH Sessions with PuTTY

In the Sessions PuTTY screen, go to *Host Name (or IP Address)*. Enter the hostname or IP address of your SSH server. You can also change the port number here if needed. Click *Open* at the bottom of the window.

The PuTTY configuration window will disappear, replaced by a black terminal window.

Saving PuTTY Connections

You can preconfigure PuTTY connections, ensuring that your sessions with a particular host happen the same way every time. Enter the SSH server's hostname under *Host Name (or IP Address)*. Under *Saved Sessions*, type a name for this connection. I usually name my connections after the host, plus possibly a word or two for any special configuration in the session. I might have a connection labeled *dns1*, and another named *dns1 with X*, so that I can easily use X forwarding when I need it.

To run a saved connection, double-click the connection name.

To copy a saved connection, highlight it, click *Load*, make your changes, and save it under a different name.

PuTTY Management

You'll see a PuTTY icon in the upper left-hand corner of your work running PuTTY session. This leads to a drop-down menu of useful tasks.

To duplicate an existing session, opening a second window to the same host, select *Duplicate Session*.

To open a new window to a host that you've already save the configuration for, select *Saved Sessions* and the session name.

To open a window to a completely new host, select *New Session*.

PuTTY Copy and Paste

PuTTY does not use the standard Windows cut-and-paste shortcuts. It works more like a UNIX-style X terminal. To copy text in a PuTTY window, highlight it with the mouse. To paste text into a PuTTY window, click the right mouse button. You can also use `SHIFT-INSERT`.

PuTTY Configuration

PuTTY stores its configuration and host key cache in the Windows Registry, under `HKEY_CURRENT_USER\Software\SimonTatham`. To move your PuTTY configuration from one host to another, copy this section of the registry to your new machine. Some people even use these registry settings to distribute valid PuTTY configurations to their users via Active Directory.

Debugging PuTTY

PuTTY has two debugging facilities: the Event Log and the session log.

The Event Log records what happens during the current SSH session. You can see the name, IP address, and port you're connected to, selected encryption algorithms, and all the various negotiations required to establish an SSH session. To view the Event Log, click the upper left corner of your PuTTY window and go down to Event Log.

For serious debugging, use a session log. Before opening your SSH connection, take a look at the left-hand pane. Under *Session* you'll find the *Logging* option. Choose it. This window gives you several options for logging your SSH session. I usually choose *All session output*. Give PuTTY a name for the log file, and browse to select a directory. Once your session has been running a while, this file will contain a large amount of detail about the session, much like the OpenSSH clients debugging option.

Changing Live PuTTY Sessions

You can alter some of the settings in an existing PuTTY session. The username and encryption information are set at login, but you can change logging, terminal behavior, and tunnel settings.

Go to the upper left-hand corner of your existing PuTTY session, and click the PuTTY icon. From the drop-down menu, choose *Change Settings*. This brings up a simplified New Session window, presenting only the options that you may change. Once you make your edits, and have confirmed that the session works the way you desire, you can save the session, either overwriting the existing name or choosing a new name.

Multiplexing Connections

SSH sessions can take a long time to open, particularly if the SSH server can't find a reverse DNS entry for the client's IP address. Or you might have a naïve firewall that limits the number of simultaneous connections between network segments. Perhaps one of the machines is so old that the initial key exchange takes several seconds. SSH supports connection multiplexing for these situations, permitting you to run several SSH sessions over one TCP connection. While this doesn't get rid of the delay for the first connection, additional sessions start much more quickly.

PuTTY supports and uses connection multiplexing by default. OpenSSH can multiplex connections, but requires additional configuration.

Configuring Multiplexing

OpenSSH's `ssh` client uses UNIX sockets to manage multiplex connections. The user must create a directory for the sockets and set the permissions so that only she can read them.

```
$ cd $HOME/.ssh
$ mkdir sockets
$ chmod 700 sockets/
```

You can now enable multiplexing in *`ssh_config`*.

```
ControlMaster auto
ControlPath ~/.ssh/sockets/%r@%h:%p The ControlMaster setting tells ssh to try to use
connection multiplexing, but to fall back to a separate TCP connection should multiplexing
fail. This lets you enable multiplexing as a default, but still connect to non-OpenSSH
servers.
```

ControlPath tells `ssh` where to find the multiplexing management files. This statement accepts tokens, much like *`sshd_config`*. The `%u` macro expands to the username, `%h` to the host, and `%p` to the port. If I connect to the host **`avarice`** on port 2222 as the user mwl, SSH automatically creates the socket file *`mwl@avarice:2222`* in the specified directory.

PuTTY enables connection multiplexing by default. To turn it off before opening a session, open PuTTY and select SSH from the left-hand pane. You'll see a checkbox called *Share SSH connections if possible*. Unselect it.

Risks of Multiplexing

Anyone who can read OpenSSH's multiplexing control files or access PuTTY's similar sockets can access all data going over your SSH connection. The original connection has already authenticated, so such an intruder wouldn't even need your password to get a terminal on the remote machine. Only use connection multiplexing on clients where you trust everybody who has administrative access.

Copying a large file over a multiplex SSH session can slow down your other sessions.

X forwarding does not work well with connection multiplexing.

Remember that when multiplexing, all of your SSH connections to a server run over the first connection you opened to the host. If that connection fails, all connections multiplexed with it will also fail.

Personally, I only enable multiplexing on single-user desktop systems. Others disagree with me. Do what makes sense for your environment.

SSH Compression

You'll hear in many places that SSH can compress data before sending it over the network. This was very useful back when a 33.6 modem was the standard

way for people to connect from home. On modern multi-megabit connections, compression normally slows down connections. Consider using compression if and only if you are seriously bandwidth-constrained.

The one case where compression makes sense is in forwarding X (Chapter 8). Adding the -C flag to ssh(1) can as much as double throughput of forwarded X connections.

That covers the basics of PuTTY and the OpenSSH client. Now let’s look at using SSH to move files around the network.

---

[1] But use one of your hosts. If you connect to mine, I might post your username, IP, and password on social media.

Chapter 6: Copying Files over SSH

File Transfer Protocol (FTP) was the standard method for copying files between machine for decades, predating even TCP/IP. FTP transmits everything unencrypted, making it roughly as secure as telnet. The file can be viewed or altered during transmission. Other old protocols, (RCP), are even worse. How about using SSH to securely transfer files between machines?

There are many ways to use SSH to move files. Applications such as rsync can use SSH as a transport mechanism. Some window managers include SSH file transfer tools. We'll cover two specific protocols, SCP and SFTP, for both Unix-like and Microsoft systems. Most other tools that transfer files over SSH are actually front ends to one of these protocols.

Secure Copy Protocol, or SCP, was designed as a drop-in replacement for RCP. SSH File Transfer Protocol, or SFTP, was designed to replace FTP. It's an interactive protocol, allowing you to browse remote filesystems. OpenSSH includes the client programs sftp(1) and scp(1), while Windows clients can use WinSCP for both protocols.

File Copy with OpenSSH

OpenSSH includes two file transfer programs, `scp` and `sftp`. We'll start with the simpler but less flexible program.

scp(1)

You can use scp(1) to copy individual files. The syntax follows the usual Unix semantics.

```
$ scp what-you-have where-you-want-it
```

Separate hosts and filenames with a colon, like so.

```
$ scp source-host:filename destination-host:filename
```

Once you authenticate, `scp` transfers the file over the encrypted channel.

If you don't enter an element in the command, it's assumed to be unchanged. For example, to copy the local file *`data.txt`* to the server **`sloth`**, run: $ **scp data.txt sloth:**

I don't enter a machine name in the source side, so it's assumed to be the local machine. I enter a remote hostname but not a remote filename, so the filename doesn't change. My file *`data.txt`* is copied to my login directory on **`sloth`**.

If the destination file already exists, `scp` silently overwrites it. If the account lacks the privileges to overwrite the file, the copy fails. The `scp` program assumes that if you told it to overwrite an existing file, you had good reason to. For this reason, I recommend not copying files while logged in as **`root`**.

You must use a colon after a hostname. When you skip the colon, `scp` assumes that the argument is a file name. Here I skip the colon and copy the file *`data.txt`*

to the file *sloth* on the local machine.

```
$ scp data.txt sloth
```

It's a very secure local copy, at least.

To change the file name on the remote side, give a new file name.

```
$ scp data.txt sloth:stuff.txt
```

If your source file is on a different machine and you want to copy it to the local host, specify the remote hostname as the source.

```
$ scp sloth:data2.txt data2.txt
```

You can copy files to or from any location where you have sufficient privileges.

```
$ scp sloth:/var/log/messages sloth-messages
```

To recursively copy a directory to another machine, use the arguments `-rp`. Here I replicate my home directory on the remote host, overwriting any files with the same name.

```
$ scp -rp /home/mwlucas sloth:
```

The `scp` program deliberately borrows many command-line options from cp(1) and rcp(1). This is why the command line options often don't match ssh(1); it's a drop-in replacement for rcp(1), so the `rcp` flags take precedence. Still, if you have more complicated copying needs, check the documentation.

The scp(1) program is largely built out of quarter-century-old rcp(1) code. This makes adding new features difficult. While nobody's looking to actively pitch `scp` into the Dead Code Dumpster, nobody's really giving it any attention either. The program is what it is.

If you have complicated file-copying requirements, look at sftp(1).

sftp(1)

The SSH File Transfer Protocol (SFTP) is more flexible than SCP. Where SCP only copies files, SFTP permits many different file operations such as renaming and removing files, listing directories, and so on. You'll find a few different protocols named after some variant of "secure" and "FTP," so don't get confused. SFTP is not the same as FTP over SSH, nor is it FTP over SSL.

SFTP commands are deliberately copied from FTP commands, to simplify transitioning between the two. Much of your knowledge of the FTP command line applies to SFTP, but we'll go through the basics.

Open a connection with the `sftp` command and a hostname.

```
$ sftp pride
```

Once you authenticate, you'll be connected and get an SFTP prompt.

```
sftp>
```

Once you've logged in, entering a question mark or the word `help` will list all the commands the SFTP server supports. FTP users will recognize most of them.

To copy a file from your local computer to the server, use `put` and the filename.

```
sftp> put upload.txt
```

To copy a file from the server to the local computer, use `get` and the filename.

```
sftp> get download.txt
```

If your connection is interrupted before the download finishes, use `reget` to resume the download where it left off. A `reget` doesn't perform file integrity checking, but only looks at the offset.

To change the name of the file on the server use `rename`, followed by the current file name and the new file name.

```
sftp> rename data.txt old-data.txt
```

To change directories on the server, use `cd` and the directory name.

```
sftp> cd /var/log
```

To change directories on the client, use `lcd` and the directory name.

```
sftp> lcd Downloads
```

End your SFTP session with either `quit` or `exit`.

Changing Usernames and Configurations

With either `scp` or `sftp`, if you use a different account name on the remote machine, put the account name and an `@` symbol right before the server name, just as you would when connecting via `ssh`. (Old-fashioned remote copy did not support this option.) $ **scp data.txt doofus@sloth:**

The easiest way to remember this is to make an entry in *`ssh_config`*. Both file copy programs take configurations from *`ssh_config`*, so make changes there once to have them affect the whole software suite.

While both programs use command-line arguments to change how they behave, those arguments are not consistent with ssh(1). SFTP is designed to replace FTP, while SCP replaces RCP. The developers prioritized comforting migrating users over using `ssh`-style options. For example, you can change the port each of these uses with `-P` rather than the `-p` used by ssh(1). Avoid confusion: use *`ssh_config`*.

File Copy with WinSCP

The PuTTY installer ships with excellent command-line SCP and SFTP clients, but if you're running Windows, you probably want a pretty graphical interface. WinSCP is a SCP, SFTP, FTP, and WebDav client for Microsoft Windows. It switches transparently between protocols depending on what the server supports.

Grab WinSCP from https://winscp.net. While there's no fee to use WinSCP in your home or business, its license (GPLv2) restricts redistributing changed versions of the program to your customers. If you wish to include WinSCP in your own product, read the license carefully.

WinSCP comes with a standard Windows installer. The defaults are fine for most users, and include convenient features such as adding WinSCP to the right-click menu when you select a file. The installer also installs Pageant and puttygen, if you didn't install those as part of PuTTY. We'll use those in Chapter

7.

Start WinSCP and you'll see this screen.

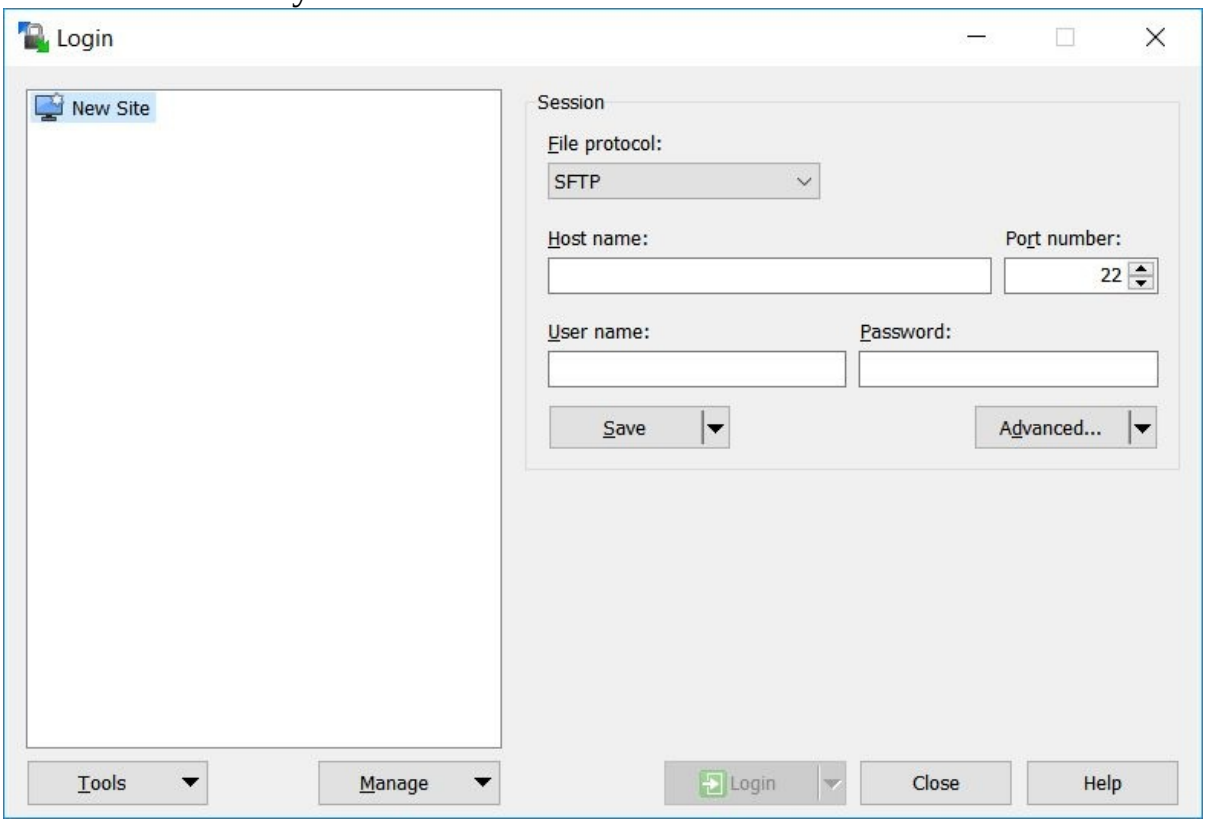


Figure 6-1: WinSCP Login The left side contains your saved connections. Set up new connections on the right. Enter the server's hostname, your username, and your password. Change the port if needed. You can save this connection by hitting *Save*.

WinSCP can import your PuTTY host key cache. Select *Tools->Import*. You'll see the contents of PuTTY's key cache, with a check box by each. Verify every server you want to use is checked, then select OK. WinSCP can now piggyback on all the work you did verifying host keys.

Once you verify a host key in PuTTY, you can go back into WinSCP and import the verified key there.

Using WinSCP

Double-click on WinSCP. Enter your username, password, and the hostname of your SSH server. WinSCP will log in and open a double window. The left side shows your local home directory, while the right side shows your home directory on the remote server. This is called a "Commander-Style" interface. Drag and drop files from one side to the other.

You can tell WinSCP to use an "Explorer-style" interface. It will open a single window, styled exactly like every other Windows Explorer window, containing the remote host. To see a local directory, you must open a separate window. To enable Explorer style, select *Tools->Preferences-> Interface* and

choose Explorer. Your WinSCP now looks so Windows-like it'll confuse even you.

And thanks to WinSCP's context menus, you can now right-click on a file and select *Send To WinSCP* to upload files.

Configuring the SCP and SFTP Servers

OpenSSH supports SCP and SFTP by default. Neither needs much configuration, but you can change a few things about how they behave.

For SCP, the scp(1) program must be in the system's default `$PATH`. If the SSH server can't find `scp`, the user will get an error saying so. On or off, present or not: that's your only option.

The `sshd` server comes with an SFTP server, activated by an *`sshd_config`* entry.

```
Subsystem sftp /usr/libexec/sftp-server The mere presence of this entry suffices to enable
SFTP support.
```

SFTP-Only Users

You probably have users that need access to copy files to or from a server, but don't need shell access. OpenSSH supports SFTP-only users. This is most commonly combined with a chroot (Chapter 3), allowing the users to access only a part of the filesystem. You'll see this in web servers that support multiple customers, where each site should be able to access only the files for their site.

Where a chrooted user who needs shell access needs a bunch of files in their chroot, an SFTP-only user needs only an *`sshd_config`* keyword.

Start by creating a group for SFTP-only users. I've called mine **`sftponly`**. By using a Match term in *`sshd_config`*, I deny these users access to anything beyond their home directory and only permit them SFTP access.

```
Match Group sftponly ChrootDirectory %h ForceCommand internal-sftp AllowTcpForwarding no
We use the ChrootDirectory keyword to lock the user in one directory. The ForceCommand
keyword restricts the user into accessing only one command. That's it! The internal SFTP
server provides all the userland commands and device node access the user might need.
```

Disabling SSH File Copy

You might want to disable the ability to copy files over SSH while still allowing users command-line access. This is really, really hard. You can remove */usr/libexec/sftp-server* and */usr/bin/scp* from your host and disable SFTP in *`sshd_config`*, but that only disables the obvious ways to copy files. Users are tricky little critters, especially frustrated users who think that the sysadmins are blocking them from doing their job. Users can copy files through any number of methods. Many of these send unencrypted data across the network. A user with shell access can always copy files from one host to another.

If you must prevent users from copying files, use chroots and limit what files the users can access. They'll still be able to copy files, but only the files in the chroot.

Chapter 7: SSH Keys

An SSH host key identifies a server. SSH also supports authenticating users with keys. Using keys to authenticate users requires more setup ahead of time than passwords, but when correctly done is both far more secure and much more convenient. We'll consider both server and user keys.

Manually Creating Server Keys

If an intruder compromises your server, the server's private key is no longer private. You must replace it. This requires generating a new key pair. While most operating systems automatically create missing host keys, others don't. Use ssh-keygen (1) to manually create server keys.

If your server runs a recent OpenSSH version, run `ssh-keygen -A` as **`root`** to automatically generate all supported but missing host keys.

You might need to create host keys for a host other than the local host, such as when deploying new installs using some orchestration systems. You can create key files manually using the `-t` and `-f` arguments to `ssh-keygen`.

```
$ ssh-keygen -t ecdsa -f ssh_host_ecdsa_key -N ''
$ ssh-keygen -t ed25519 -f ssh_host_ed25519_key -N ''
```

The `-t` flag specifies the type of key to create. Here we create two different types of keys, ECDSA, and ED25519. The `-f` flag gives the file name of the private key file. The public key for each key pair is in a file of the same name with *`.pub`* added to the end. Finally, `-N` lets you specify a passphrase on the command line. Host keys have no passphrase. The two single quotes indicate an empty passphrase.

Whenever you generate host keys, be sure to get the key fingerprints as discussed in chapter 4. Your users will need the fingerprints to verify the host keys.

Passphrases

What's this passphrase thing I just mentioned? A passphrase is like a password, but longer. It includes spaces, words, special characters, numbers, and anything else you can type. The passphrase is used to encrypt and decrypt the private key. A key with a passphrase cannot be used until someone enters the correct passphrase.

Passphrases are most often used with user authentication keys. A user with a key pair can access the system without providing a password for that system. Desktop and laptop systems are usually less secure than servers, and get infected, hijacked, or outright stolen depressingly often. If a user's authentication key pair is stolen, the intruder can use that key pair to access servers just as if he

was the legitimate user. Encrypting the private key with the passphrase means that even if the user's private key file is stolen, the intruder cannot use the key without the passphrase. If an intruder gets either your private key file or your passphrase, but not both, the damage is contained. Make the passphrase too long to guess by brute force and sufficiently complex to discourage casual eavesdropping.

Can a passphrase be a single word, like a password? Yes, but it's a *really* bad idea. Computers are now so fast that they can quickly discover short passwords by trying all possible passwords in succession. Using a short passphrase considerably reduces your private key's security.

A passphrase should be at least several words long, something you can easily remember, and shouldn't be obvious to others—even to people who know you. It should include special characters such as #, !, ~, and so on. Peculiar words from specialized non-computing vocabularies are useful. Substitute numbers for letters. Never use anything from pop culture, and never use any of your own personal catchphrases. Anything you've said to friends or coworkers that was catchy enough to repeat is a poor choice. If your imagination completely fails, Diceware (http://www.diceware.com) is a tool for randomly generating mostly-memorable passphrases from real words using ordinary dice. While intruders can ruin your week, a coworker with your private key and a sense of humor can be even more aggravating.

Host keys do not use passphrases, because the SSH service must start when the system boots. You could use a passphrase with the server key, but SSH would not start until someone entered the passphrase at the server console. This is unacceptable in most environments.

User Keys

User key pairs provide stronger authentication than passwords. Combined with agents (see "SSH Agents" later this chapter), user keys eliminate the need to type any authentication credentials into remote machines. Cryptographically, user keys are identical to host keys. The only difference is where the keys are used.

Speaking very generally, a computer can identify you based on something you are, something you know, or something you have. Iris scanners and fingerprint readers verify your physical body, something you *are*. A password verifies that you *know* a secret. Getting into a house requires that you *have* the door key. Key-based authentication combines two of these: you must *have* the file containing the private key and you must *know* the passphrase for that key. Admittedly, a private key file is easier to reproduce than a physical key—it's

only copying the file—but it's more difficult to reproduce than an 8-character password. This additional layer of security provides extra protection against unauthorized use of an account.

Keys are more complicated than passwords, however. Just as you wouldn't leave your front door key hanging from the doorknob, you must protect your private keys. If the computer is lost or stolen, any private keys on that machine should be considered lost as well. While it's possible to remember a password, most people won't put in the time or energy to remember the thousands of characters in a private key. Yes, you should have backups… but if your laptop is stolen, the private keys on that laptop should be considered compromised anyway.

Is setting up authentication via user keys really worth the trouble? For almost a decade, a network of compromised machines dubbed the "Hail Mary Cloud" has repeatedly scanned the Internet for SSH servers. When a cloud member finds an SSH server, it lets the other machines in the network know about it. The cloud then methodically tries possible usernames and passwords. One host on the network tries a few times, then another, then another. Blocking individual IP addresses is not a useful defense against these scanners, because each address only tries a few passwords before the next attacker takes its turn.

Any one attempt has low odds of guessing successfully. The attempts are constant. They never end. Eventually the Hail Mary Cloud will get lucky and break into your server. It might be tomorrow, or next year, but it *will* happen. To prevent this intrusion, you can either use packet filtering to block public access to your SSH server, or you can eliminate password authentication. User keys let you eliminate passwords.

SSH Agents

Replacing a password with a passphrase and a private key has one obvious flaw: typing passwords is an annoyance. Why replace an annoying password with an even more annoying passphrase? It might be more secure, but are you and your users really going to bother?

That's where an SSH agent comes in. An SSH agent is a small program that runs in the background. When you start a desktop session, you enter your passphrase to decrypt your private key. The decrypted private key is loaded into the SSH agent. The agent stores the key in memory, never on disk. The agent processes all private key operations for the SSH client. When the SSH client needs to decrypt something with the private key, it asks the agent to handle it. When you log off for the day, the SSH agent shuts down. The decrypted private key disappears from memory. In other words, with an SSH agent, you type your

passphrase once per work session, no matter how many SSH sessions you open that day.

On a typical day I log into my workstation, activate my SSH agent, and type my passphrase once. I then open innumerable SSH sessions to servers and routers all over my network, without typing a passphrase or password again. When I log off for the day, my agent shuts down. The memory used by the agent is wiped and returned to the operating system. My private key is once again available only in the encrypted file.

Agents do not guarantee security. Anyone who can read your computer's memory while you are logged in can access the decrypted key. This includes the `root` account. If you don't trust the system administrator on your desktop, don't use an SSH agent.[1] If you suspend your laptop, the decrypted private key remains in memory. Anyone who can wake your laptop and login can use the key as their access rights permit. A random thief interested in swapping your laptop for a quick buck probably won't know or understand what he has, but a thief who is specifically targeting you and/or your employer will probably check for a live private key. More commonly, if you don't lock your desktop before going to lunch, a coworker might take advantage of your unsecured terminals. These problems are best solved by emptying or shutting down your agent when you're not actively using the system.

Agent security is also a problem on multiuser machines. Anyone who has administrative or superuser privileges on the system can access any SSH agent running on the host. If other people have root or Administrator access on your desktop, they can access your agent and masquerade as you. Using an agent would be unwise.

We'll discuss SSH agents for both PuTTY and OpenSSH later this chapter.

Creating an OpenSSH User Key

If you have a Unix-like desktop, generate a key using ssh-keygen(1). Don't use any arguments and the program will walk you through generating a user authentication key.

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/mwlucas/.ssh/id_rsa): Enter passphrase (empty
for no passphrase): Enter same passphrase again:
Your identification has been saved in /home/mwlucas/.ssh/id_rsa.
Your public key has been saved in /home/mwlucas/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:LK+1dKbb/PtN8KjiXLDdZzOK1fivRkMZIn8YoOrmEvA mwlucas@zfs1
The key's randomart image is:
+---[RSA 2048]----+
…
```

You'll be asked where the new key should be saved. The various OpenSSH programs expect to find key files in the default locations, so take the suggestion.

You'll then be asked to enter a passphrase twice, to verify that you can type it more than once. Your private key will be encrypted with this passphrase. Always use a passphrase, as discussed just a few pages previous.

SSH uses identical key formats for hosts and users. When you generate a user key, you get a key fingerprint and a randomart image. Neither is particularly useful for user authentication keys.

You'll find your new private key in *`$HOME/.ssh/id_rsa`* and your new public key in *`$HOME/.ssh/id_rsa.pub`*. Immediately backup your new key pair on off-line media, such as a flash drive or CD-ROM. If you destroy your workstation, you'll want the ability to recover your key pair.

Key Algorithms

Like host keys, user keys can use a few different encryption algorithms. If you don't specify an algorithm, the OpenSSH tools use the recommended one—at this time, 2048-bit RSA. You can specify a different algorithm with the `-t` flag.

```
$ ssh-keygen -t ecdsa
```

Why create multiple keys? Cryptographers have this distressing habit of finding weaknesses in cryptographic algorithms. One day the unthinkable will happen and someone will discover a flaw in a widely used and broadly trusted algorithm. All keys that use that algorithm will immediately become untrustworthy. If you have user keys with different algorithms, you can disable the broken algorithm on your SSH server and still have server access.

Our examples assume that you're using an RSA key, but they're just as applicable for keys made with other algorithms.

Creating a PuTTY User Key

Use the PuTTYgen program to create user authentication keys for PuTTY. The PuTTY installer includes puttygen, or you can download it individually from the PuTTY website. When you start puttygen, you'll get a screen like Figure 7-1.

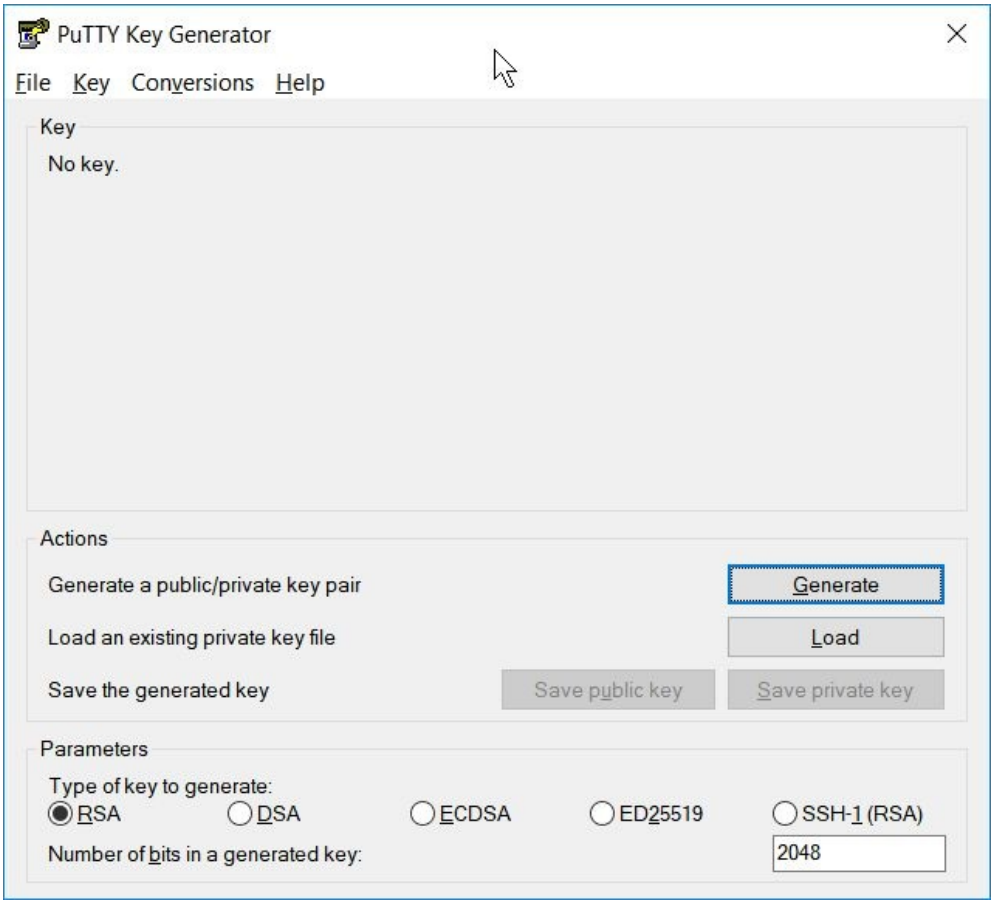


Figure 7-1: PuTTYgen Startup Use RSA, ECDSA, or ED25519 keys. DSA keys are on their way out, and SSH-1 RSA keys are far obsolete. Verify that the number of bits is at least 2048. More is not necessarily helpful. You might use fewer bits for user keys dedicated to ancient servers, such as VAXes or Alphas. Click *Generate*. The next PuTTYgen screen asks you to generate randomness by wiggling the mouse over the blank area. Once you generate sufficient entropy, PuTTYgen creates your key, as displayed in Figure 7-2.

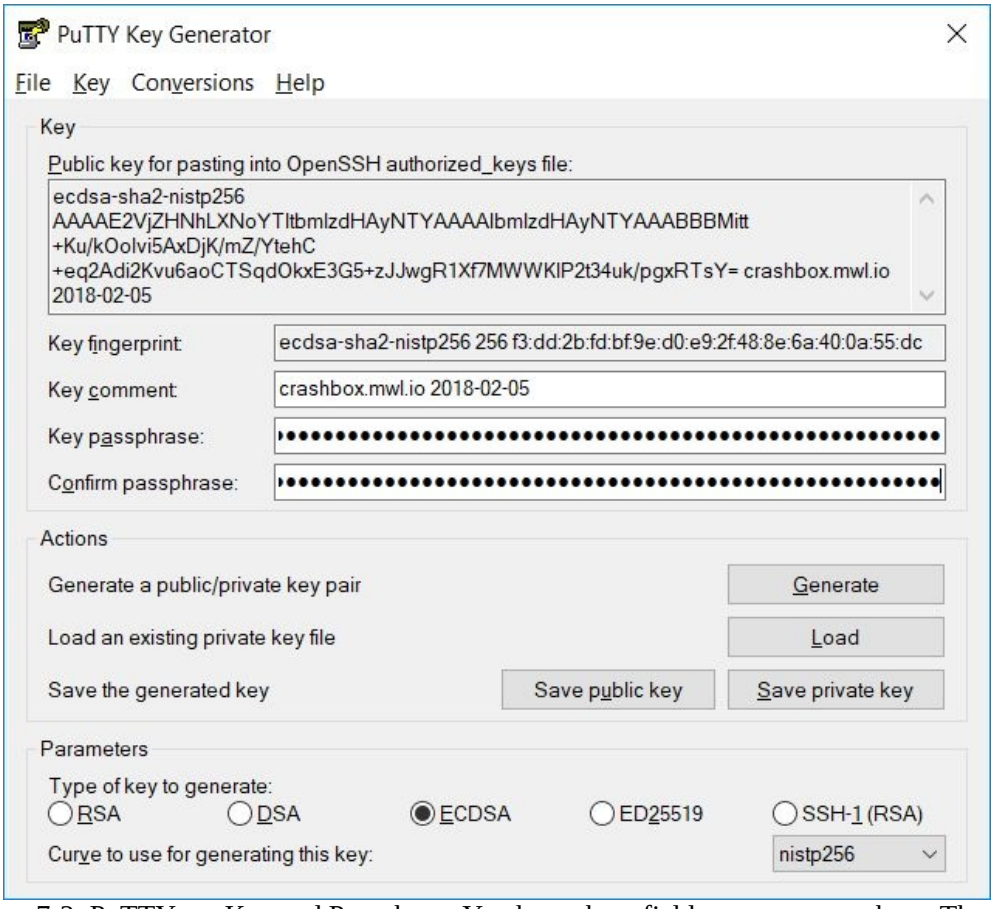


Figure 7-2: PuTTYgen Key and Passphrase You have three fields you can enter here. The first is the key comment. I recommend changing the comment to reflect the machine you generated the key on and the date you created it. Now enter your passphrase twice.

Now click `Save public key`. You'll get a standard Windows Save as dialog box asking you to choose a location to save the key. Save the file in a location that only you have permissions to access. You can use a folder under Documents, but make sure you go in later and set the permissions so that other users on your machine cannot view the file. I normally name these files after the machine they're created on and the date. Puttygen won't assign a file extension by default, so use a .pub extension.[2]

Now save the private key. Use the same file name for the public and private keys. PuTTYgen uses a .ppk extension for private keys, so they won't overwrite each other.

You now have a public key. Congratulations! But don't exit PuTTYgen yet.

The top of the screen shows the key in OpenSSH-friendly *`authorized_keys`* format. Copy that into a file, all on a single line. I usually name that file after the machine the key was generated on, the date, and add the string *`authorized_keys`*.

Installing Public Keys

No matter which client you use, you must install your public key on your server

before you can login with it. Whenever you SSH into a host, the OpenSSH server checks the local file *$HOME/.ssh/authorized_keys*. This file contains public keys, one per (very long) line. The SSH server compares the public key offered by the client with the keys in the file. If the key matches, and the client can successfully exchange data with that key, then the client has demonstrated it has the corresponding private key. Access is granted. If there is no *authorized_keys* file, the server falls back to the next authentication method (usually passwords).

If you are a user requesting access to a server that only accepts public key authentication, the sysadmin will ask you for your *authorized_keys* file. If this is your first time using public key authentication, this is not a security risk—remember, your public key file is public. Anyone can have it. It's utterly useless without the corresponding private key.

The most common type of user key is an RSA key. OpenSSH stores your client's RSA public key in the file *$HOME/.ssh/id_rsa.pub*. PuTTY's key generator makes you name your own key files, and you'll have a few different key files. You want the file containing the authorized_keys-friendly version. If you followed my suggestion, the file name will contain *authorized_keys*. To simplify the examples, we'll use the file name *id_rsa_authorized_keys.pub*. Substitute your PuTTY file as needed.

To use your public key, you must copy the client's public key file to the *authorized_keys* file in your account on the server. You could use the graphic interface's copy and paste function, but that's error-prone. Uploading the public key file via SFTP or SCP and then concatenating it onto *authorized_keys* is more reliable. Remember, each key must be on one and only one line in *authorized_keys*. More than one of my simple cut-and-paste attempts have turned to tears, then to threats of starting a new career as a llama smuggler, only to end in a manic-depressive binge at the nearest gelato shop. Have the machine copy the file. It's better at it than you are.

Here, I copy my client's *id_rsa_authorized_keys.pub* to the server **sloth** using scp(1).

```
$ scp .ssh/id_rsa_authorized_keys.pub sloth:
```

The server will still request a password to upload the key file; you've created the key, but it's not yet installed.

PuTTY users should use WinSCP's friendly drag-and-drop file copy.

Now log onto the server and append the contents of *id_rsa_authorized_keys.pub* to the *authorized_keys* file. If this is the first time you've installed a public key, you could copy your key file to *authorized_keys*. If you let yourself get into that habit, however, one day you'll overwrite an existing *authorized_keys* and spend the next couple of hours kicking yourself for making such a simple mistake.

```
$ cat id_rsa_authorized_keys.pub >> .ssh/authorized_keys
```

Now you can try to authenticate with your key. If key-based authentication doesn't work for you, check the permissions on *authorized_keys* and the *.ssh* directory. Neither should be writable by any user except you.

If you are uploading from a UNIX-like host, you can do the upload and copy in one command.

```
$ cat .ssh/id_rsa_authorized_keys.pub | ssh sloth "cat >>~/.ssh/authorized_keys"
```

If you ever manually edit *authorized_keys*, be certain that the last key ends with a newline. If the final entry doesn't end in a newline, the next key you add to this file will be tacked onto the end of the previous key. Both the new key and the old key will stop working. If in doubt, go to the end of the file and hit RETURN. An extra newline at the end of *authorized_keys* won't hurt anything.

Only upload the public key, *never* the private key. Your private key should never cross the network.

Once you have your *authorized_keys* file exactly the way you want it, you'll want to copy it to all of your servers. There's lots of ways to manage this. Many UNIX-like hosts include ssh-copy-id(1), a convenient way to copy an existing *authorized_keys* from one host to another. I have my up-to-date *authorized_keys* stashed on a public Web server, so that I can easily install it on any machine I happen to wander into. Or, you can use the techniques discussed in Chapter 11 to automatically copy the *authorized_keys* files for you and all of your users to all of your machines.

Now test your key from your client.

Authenticating with Keys

Using a key for authentication changes how you log in. No matter what client you're using, verify that your key works before going any further. Don't attempt to use an SSH agent until you know the key works.

If the key doesn't work, use the SSH debugging tools discussed in Chapter 5. Run ssh in verbose mode. Use PuTTY session logging. Read the output. If you have a permissions problem or configuration error, the answer is in there.

Using OpenSSH User Keys

When your client finds a key pair in *$HOME/.ssh* and the SSH server finds an *authorized_keys* file in your account, the client asks you to enter your passphrase. Here I connect to the remote machine **sloth**: $ **ssh sloth**

```
Enter passphrase for key '/home/mwl/.ssh/id_rsa': All of the software involved has found
your key files. Once you enter your passphrase, the client can decrypt the private key and
use it to authenticate with the server.
```

Yes, this looks much like a regular password-based logon, but behind the scenes it's very different. You've decrypted the key file locally. The only authentication information you've sent to the server is confirmation that you're

able to exchange data encrypted with a public key stored in the *authorized_keys* file in your account on the server. You never send a password or other traditional authentication information.

OpenSSH automatically checks for all the standard key files. You might have a special-purpose key that's only used for special circumstances, such as automated jobs. To use that key with `scp`, `sftp`, or `ssh`, use the `-i` flag and the filename.

```
$ ssh -i $HOME/specialkey sloth
```

Now that you know your key works, you'll need to enter your passphrase every time you log onto this server. This is a good time to configure an SSH agent.

Using PuTTY User Keys

If you don't have an agent running, you must tell PuTTY where to find your private key file. On the left side of the PuTTY Configuration screen, select *Connection-> SSH-> Auth*. In the text box labeled `Private key file for authentication`:, put the full path to your private key file. Remember, the private key file ends in .ppk.

Now try to connect. PuTTY should prompt you for your username and then request your passphrase. If you enter the passphrase correctly, you'll get a command prompt.

Once you know that your key works and is installed correctly, reduce how often you must type your passphrase with the Pageant SSH agent.

SSH Agents

While the SSH agents for OpenSSH and PuTTY are wildly different, both perform identical tasks. They host your private key in secure memory so that you don't have to keep typing your passphrase. Both let you view the decrypted keys, add new keys, and delete keys. The only real difference between them is how they're programmed and how you make them behave—you know, the unimportant stuff.

With your key loaded into `ssh-agent`, your login attempts will look like this: $ **ssh mail**

```
Last login: Thu Nov 16 16:56:52 2017 from ceo.worldhq.mwl.io FreeBSD 10.3-RELEASE-p20
(GENERIC) #0: Wed Jul 12 03:13:07 UTC 2017
```

Note the absence of any request for a password or passphrase; you've just logged into the remote machine without human authentication. If you connect to many machines during your working day, an SSH agent makes life much easier and transforms user keys from an annoyance into a pleasure.

OpenSSH Agent

Any UNIX-like system that includes OpenSSH has the SSH agent ssh-agent(1).

And that's where the easy stuff stops.

One annoyance about the multiplicity of desktop environments in the UNIX-like world is that every environment has its own preferred way of running `ssh-agent`. We'll discuss a couple of them here, but if none of these work in your environment, you'll need to check your operating system or window manager documentation. Many Unix variants have their own slightly unique desktop setups, and they change the precise methods of using `ssh-agent` to suit the developers' personal prejudices and the Whim Of The Week.

Most display managers, like `xdm` and `kdm`, have hooks to automatically check for SSH keys in the user's home directory. When the display manager finds a key during the logon process, it creates a pop-up window to request your passphrase. Enter the passphrase and the display manager attaches the SSH agent to your desktop environment. You're ready to begin work.

If you're more old-fashioned and run your desktop with startx(1), tell the SSH agent you have a key with ssh-add(1) before running `startx`.

```
$ ssh-add
Enter passphrase for /home/mwl/.ssh/id_rsa: Enter your passphrase to add your keys to the
agent.
```

Text console users must first run ssh-agent(1) with their shell as an argument, and then run `ssh-add`.

```
$ ssh-agent /bin/tcsh
$ ssh-add
```

All SSH sessions that start from that console session run with the agent. The agent doesn't work across virtual console terminals, only for the children of the shell run by `ssh-agent`. Another virtual terminal needs its own SSH agent.

If you have multiple keys with the same passphrase, `ssh-add` automatically decrypts all of the keys. If you have multiple keys with different passphrases, `ssh-add` prompts for each passphrase separately.

Use `ssh-add -l` to list all private keys currently stored in the agent, and `ssh-add -D` to remove the keys from a running agent. Re-add them once you get back from lunch.

PuTTY Agent

The PuTTY SSH agent, Pageant, provides a friendly Windows-style interface to SSH. Start Pageant by double-clicking on it. The Pageant icon, a computer with a black broad-brimmed hat, will appear in the system tray.

Right-click on the Pageant icon. You'll see several options, including *View Keys*, *Add Key*, and *Exit*. There are also options for running a saved or new PuTTY session. Select *Add Key* to bring up a standard Window file browser. Find your private key and select it. Pageant will display a dialog box to request your passphrase. Enter it. If you can't type your passphrase correctly, Pageant

will ask you to do it again.

Once Pageant is ready, open a PuTTY session. Connect to a machine that has your public key installed. You should get a command prompt without needing to enter your passphrase.

If key-based authentication works when you specify a private key file, but not when using Pageant, verify that PuTTY is configured to use Pageant. Select *Connection->SSH->Auth*. Under *Authentication Methods*, you'll find an *Attempt authentication using Pageant* checkbox. Make sure it's checked.

Once you know that Pageant works, it's helpful to have it start at login. Find your account's Startup folder (the exact location varies depending on your version of Windows). In another window, find your Pageant program. It will probably be in the PuTTY directory under either `Programs` or `Program Files (x86)`. Create a link to Pageant in the startup directory.

For optimal convenience, have the shortcut load your private key at login. You can either give Pageant the full path to your key as an argument, or you can set a *Start in* directory and only use the short file name. I recommend setting the *Start in* directory, because it makes loading multiple keys at login much simpler. Right-click on your Pageant shortcut. Under *Target*, add the name of your private key file as an argument. The Target should now look something like `"C:\Program Files\PuTTY\pageant.exe" moose.ppk`. On that same screen, you'll see a *Start in* field. Enter the full path to your keys directory there. On my laptop, that would be `C:\Users\mwlucas\Documents\keys`.

To verify this works, exit your running Pageant and double-click on the new icon in the startup folder. You should be prompted to enter the passphrase for your key. If it doesn't work, you've probably messed up a path in the shortcut. Remember that you need to put quotes around any path with a space in it.

Backing Up Key Files

If you lose your private key, your key pair is useless. Once you know your key pair works for authentication, back up both the private and public keys. The PuTTY .ppk file contains both the public and private keys, but OpenSSH key pairs need both files. Don't just copy your private key to another machine—every machine that has your private key is another place your key can be stolen from. Back up your private key on off-line media, such as a flash drive or a CD. You might also encrypt it with a program like GnuPG. (If you are not familiar with GnuPG, I recommend the book *PGP & GPG: Email for the Practical Paranoid*, by yours truly.)

Keys and Multiple Machines

Many sysadmins have multiple computers. I regularly use two desktops and a

laptop. It is possible to move key pairs between machines by copying the key files. You can even import OpenSSH keys into PuTTY. How do you realistically manage a single key between multiple machines?

You don't.

Rather than reusing a single private key on all of your desktops and laptops, create a separate private key for each. Create an `authorized_keys` file that contains the public keys for all of your authentication keys. When a machine is decommissioned, stolen, or self-immolates, remove that machine's key from use. Delete the corresponding public key from the `authorized_keys` file on all of your servers. Generate a new key on your replacement machine.

If one of your desktop machines is compromised, you must remove that machine's authentication key from use. If all your clients share a single private key, you must regenerate a new key pair and distribute it to all of your machines. The intruder who has your private key might well lock you out of your own systems before you can accomplish this. If each machine has a unique key pair—even if all the keys share the same passphrase—then compromise or loss of one key does not compromise the keys on all your other machines.

Also, preferred key algorithms change over time. When I wrote the first edition of this book, user authentication keys defaulted to 1024 bits. The default is now 2048. If I was still using keys created years ago, they would be too weak for current use. It's entirely possible that the default user authentication key algorithm of RSA will be replaced by an entirely different algorithm in the future. By creating a new key whenever you get a new machine, and invalidating keys associated with old hardware, you ensure that your keys are relatively recent and secure.

Disabling Passwords in the SSH Server

Passwords are less secure than keys. Now that you have working key-based authentication, the smart thing to do is to disallow password-based authentication. The `sshd_config` keyword ChallengeResponseAuthentication disables generic challenge-response authentication systems, such as a prompt requesting a username and password. The keyword PasswordAuthentication enables and disables passwords. To disable password-based authentication, set both of these keywords to `no`.

```
ChallengeResponseAuthentication no PasswordAuthentication no
```

While `sshd` permits public key authentication by default, verify that nobody's changed that keyword. The PubkeyAuthentication keyword must be set to `yes`.

```
PubkeyAuthentication yes
```

Now restart `sshd`, either with the built-in system command or `pkill -1 sshd`.

Changing `sshd_config` will not change how other programs use passwords. If

you use passwords for `sudo`, `sudo` will still ask users for their password.

Password Authentication Warning!

If you make a mistake in configuring SSH such that nobody can login, you can lock yourself out of your server. When making changes to *`sshd_config`*, do not log out of your existing SSH session until you verify your changes work. (Remember, you can run `sshd` on an alternate port for testing, as discussed in Chapter 3.) Create a new SSH session. Verify that you can login and become **`root`** before disconnecting your first session.

The preceding paragraph is very important. Ignore it at your peril, or be prepared for your own manic-depressive gelato binge.

Permitting Passwords from Specific Hosts

While passwords are weak, sometimes you cannot disable them entirely. You might have a few users or applications that cannot use keys for one daft reason or another. The underlying problems are most often political rather than technical, but they're still problems. While you're working on solving those problems, you can allow passwords from specific hosts with conditional configuration, as discussed in Chapter 3. Here, we allow password authentication from a particular subnet.

```
Match Address 192.0.2.0/24
PasswordAuthentication yes
```

Remember, all Match statements go at the bottom of *`sshd_config`*.

It is possible but extremely unwise to permit password authentication based on username. The SSH server, rather than hanging up on clients that request passwords, must get the username before hanging up on the client. This means that the Hail Mary Cloud will continuously poke at the server. The reasons that compel you to permit limited password authentication probably make requiring a strong password just as problematic—the same boss that demands he be allowed to use passwords probably thinks that p455w0rd is a secure password. The account that permits passwords will be a weak spot. The only thing that can save you here is good off-host logging to a very secure bastion host, so that when the machine is compromised you can tell the boss that the downtime is his fault.

Agent Forwarding

Suppose I disabled password-based authentication on all my computers. The only way to access a command prompt on any of my hosts is by authentication with public keys. I'm working on my server **`wrath`**, and must copy a file over to the server **`gluttony`**. This presents a problem. My private key isn't on **`wrath`**. Copying the private key to a server is terrible security practice—you want your private key on as few hosts as possible, and never on your servers. But passwords don't work. How can I use SCP or SFTP?

The answer is to forward authentication requests back to your workstation. Agent forwarding is exactly that. When you try to SSH from one server to another, the SSH client on the server sends private key requests back to your desktop. The agent is available as a socket, in a location given by the environment variable `$SSH_AUTH_SOCK`.

```
$ echo $SSH_AUTH_SOCK
/tmp/ssh-zOeUnDTnkb/agent.2513
```

To use agent forwarding, both the client and the server must permit it and the SSH agent must be running before starting your first SSH connection. If both the client and the server support and request forwarding, the authentication request will be forwarded.

Agent Forwarding Security

The risk of agent forwarding is that you must extend some trust to the SSH servers. Anyone who has root access can access your SSH agent socket. Anyone who can access your SSH agent's socket can use your private key without providing a passphrase.

If your SSH server is compromised, the intruder can piggyback onto your authentication socket to log into remote servers with your credentials. Promiscuous agent forwarding has been responsible for intrusions in many organizations, even organizations you'd think would know better. Only enable agent forwarding to machines that you control.

Agent Forwarding in sshd

To enable agent forwarding on the server, set to the AllowAgentForwarding keyword to `yes`.

```
AllowAgentForwarding yes
```

I'll generally disable agent forwarding globally, then use a Match statement to permit only certain users or addresses to forward their agents.

OpenSSH Client Agent Forwarding

In *`ssh_config`*, use the ForwardAgent keyword to activate agent forwarding.

```
ForwardAgent yes
```

The next time you connect to a server, the client will request agent forwarding.

PuTTY Agent Forwarding

PuTTY enables agent forwarding by default. On the left side of your PuTTY setup screen, go to *Connection -> SSH -> Auth*. Under *Authentication parameters*, you'll see a check box labeled *Allow agent forwarding*.

You can also use key authentication and *`authorized_keys`* to very specifically restrict what a user may do over SSH. We'll examine that in chapter 12. Now, let's look at forwarding X.

[1] Not trusting the desktop's sysadmins basically destroys any hopes of server security. Yes, we've all worked there.

[2] Don't open public keys in Microsoft Publisher. That doesn't make anybody happy.

Chapter 8: X Forwarding

Unix-like systems use the X protocol (or X11) to display a graphic user interface. X has improved over the years but it's still famously baroque. One of X's more useful features is the separation between the system a program runs on and the system the program's display appears on. You can run a program on one host, and have the display appear on a completely different workstation. I can run a graphical web browser on a host on the public Internet, and funnel the display back to my laptop inside my employer's firewall, bypassing the firewall content filter restrictions—for completely legitimate work reasons, of course. In this scenario all web requests originate from my server, and the results appear on my laptop. You can do the same thing with any X program.

If you've never used X before, it might seem a little strange. That's okay. Play with it and you'll quickly understand its usefulness.

Vanilla X transmits information across the network unencrypted. Secure X in-transit by wrapping it with SSH via X11 forwarding.

X Security

X dates from a time when network security was not nearly the issue it is today. The developers were happy to get graphic applications working at all, given the limited hardware available in those days. Retrofitting security into any protocol isn't as effective as we might hope. Displaying X from a remote machine requires extending trust to the remote machine. The more you trust the remote machine, the more X programs you can display locally. If you fully trust a compromised machine, the intruder can use X to take over your workstation, capture your keystrokes, and access your systems as if she was you.

Only permit X forwarding to users or hosts that truly require it.

X and the Network

Back when X was developed, a site's Internet connection might be as fast as 56 kbs. Attempting to use X forwarding between sites was the sort of things sysadmins would laugh at over a beer. Now that bandwidth is not such a concern, though, people might run a browser on one continent and display it in another.

While bandwidth is no longer an issue in many parts of the world, latency is very real. Many graphical programs are highly sensitive to latency. A program might reasonably expect to perform several hundred graphics operations a second. That's fine when each takes a nanosecond. When each takes fifty milliseconds thanks to the cross-country link, though, your program becomes unusable. If latency isn't a problem, jitter and packet loss can destroy usefulness.

If you have any latency at all, investigate alternatives to forwarding X. Maybe dynamic port forwarding (Chapter 9) would solve your problem. Perhaps your program has a feature for remote use, such as Wireshark's ability accept a `tcpdump` stream from another host. Use a protocol designed to accommodate high latency.

Inside your local network, though, X forwarding can be incredibly useful.

The X Server and Client

The X server is the computer that provides the graphic display. The X client runs the program that generates the display. This seems backwards to many people. If you are using X, the X server is almost certainly your desktop. Your desktop must have an X server to use X forwarding.

Almost all Unix-like systems include an X server, usually from X.org but possibly a vendor's proprietary system. If you are running Windows, you'll need a third-party X server. We'll cover those in the discussion of PuTTY and X.

X Forwarding on the SSH Server

To use X forwarding, the SSH server must have the xauth(1) program. If it's present, you can enable forwarding with the X11Forwarding keyword in *`sshd_config`*.

```
X11Forwarding yes
```

Restart (or `pkill -1`) `sshd` after making this change.

The OpenSSH manuals mention several other options for configuring the fine details of X forwarding, but the overwhelming majority of you will never need any of them. If you have an odd problem, investigate the various X11 keywords in sshd_config(5).

X Forwarding in the OpenSSH Client

The OpenSSH client supports two levels of X forwarding, differentiated by security level. Configure both in *`ssh_config`*. Basic X forwarding supports only a less-insecure subset of the X protocol. This level of X forwarding is fairly safe. Intruders cannot take over your desktop or snoop your keystrokes with basic X forwarding.

```
ForwardX11 yes
```

Always try this basic X forwarding first.

Many X programs use functions beyond the less-insecure subset. When forwarded over SSH, these programs show an error and unceremoniously crash. Once you enable X forwarding, you can choose to allow the full set of X functions with the keyword ForwardX11Trusted.

```
ForwardX11Trusted yes When you permit all X functions, you fully trust the SSH server. An
intruder who controls the SSH server can capture everything on your local screen and your
every keystroke. Be really, really sure you trust every single remote server you might
ever log into before permitting this level of trust globally. And once you're absolutely
```

certain—don’t do it.

X forwarding is one of those rare places where SSH compression makes sense. Set the Compression keyword to `yes` to enable compression. It’s best only used on a per-host basis, however.

Per-Host X Forwarding

You can configure per–host settings to restrict X forwarding to only necessary hosts, using Match rules. Here I have a program on **`pride`** that requires fully trusting X, so I make a special entry for it in *`ssh_config`*.

```
ForwardX11 no
Host pride
ForwardX11 yes
ForwardX11Trusted yes Compression yes
```

Now I only have to worry about X software on one host, not every host I SSH into.

Forwarding X on the Command Line

Even better then restricting X forwarding to certain hosts is enabling it on a connection-by-connection basis.

In the previous example I fully trust X for all connections to the host **`pride`**. I have a program on that host that needs full X access, but I don’t run that program every time I log into that host. I want to enable X forwarding for only certain sessions. Enable compression with the `-C` flag. Activate standard X forwarding when necessary using the `-X` command-line option.

```
$ ssh -CX pride
```

If you must fully trust the remote host, equivalent to ForwardX11Trusted, use `-Y`.

```
$ ssh -CY pride
```

This eliminates the risks of routinely forwarding X, but supports X forwarding when necessary.

X Forwarding with PuTTY

The first problem with forwarding X to a Windows host is that Windows does not include an X server. You need additional software. Fortunately, many people have ported the standard X.org software from UNIX to Windows. Use any of them you like. I generally use Xming, but don’t worry if your employers or coworkers insist you use a different one.

Xming

Xming is a widely used and frequently updated X server for Windows systems. The most recent version of Xming is only available to people who donate to the project, but the next older version is free. As with all of the software in this book, if you find Xming useful, I encourage you to donate to the programmer. Xming brings to Windows all sorts of X tricks familiar to UNIX users, but for our purposes we’ll use it only to display programs running on a remote machine.

Download Xming from http://sourceforge.net/projects/xming/. The Xming installer is very straightforward to any Windows user, so I won't walk you through it. Take the defaults. Once you complete the install, run Xming to start the server.

Enabling and Disabling X Forwarding

PuTTY forwards X by default. What's more, PuTTY does no security-based filtering of X; it's forwarding is equivalent to ForwardX11Trusted in `ssh`. For this reason, I recommend disabling X forwarding by default, then enabling it only when needed.

On the left-hand side of the PuTTY Configurations screen, select *Connection -> SSH -> X11*. The first checkbox is *Enable X11 forwarding*. Deselect it, then save the Default Settings. Leave the other settings unchanged, as they're only useful in uncommon situations.

Is Forwarding Working?

Your SSH session won't look any different after you forward X. How can you prove forwarding works before you need it? If SSH has successfully negotiated X forwarding, it will set the `$DISPLAY` variable in your shell.

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

Your shell knows that there's an X server attached to it. You can run your X program. If forwarding isn't working, `$DISPLAY` is undefined. Check your system log, or the debugging log of your SSH client.

A connection using the insecure, legacy protocol XDMCP will have a `$DISPLAY` value of something like `remote:1`. This means that your shell found an X display, somewhere, somehow, but it's not the one you're trying to forward over SSH. Don't run your X program if `$DISPLAY` looks weird! Something might be very, very wrong.

Now run an X program from your shell, and it should display on your desktop. Most X clients include the xterm(1) terminal emulator. Run `xterm` in the background on your SSH server.

```
$ xterm &
```

You'll get a command prompt back on the SSH server. In a moment or two, depending on the bandwidth and latency between your server and client, a terminal on the remote system will appear on your desktop.

If you don't like `xterm`, try `xclock`, `xeyes`, or `xcalc` instead.

When you connect with X forwarding enabled, you might see warnings like `untrusted X11 forwarding set up failed` or `No xauth data`. These warnings are not critical when forwarding X over SSH, and should not worry you.

Remote X Commands with OpenSSH

Logging into another machine just to run an X program—or any program—can be an annoyance. The `-f` option to `ssh` lets you run a command on another machine while keeping the SSH session itself backgrounded. This looks like you're executing a command directly on the other host. Give the command right after the host you want to access. For example, if I want to run an xterm on `wrath` I could run: $ **ssh -f wrath xterm**

The client will connect to the server and display whatever login text the server shows. The SSH client then goes into the background, restoring your command prompt on your local system even as it runs the command on the remote system.

Note that remote commands are run in the user's full logon environment. Any files attached to the user's shell, such as *.cshrc* or *.profile*, are sourced. This might give you trouble, depending on the application you're running.

Backgrounding forwarded X-over-SSH sessions is very useful, but forwarding TCP ports over SSH is even more useful. We'll look at that next.

Chapter 9: Port Forwarding

Port forwarding over SSH is a divisive topic.

SSH can serve as a wrapper around arbitrary TCP traffic. You can cloak unencrypted services such as telnet, POP3, IMAP, or HTTP inside SSH, securely transporting these natively insecure protocols. An SSH session can carry any TCP/IP protocol, including protocols your local IT security team has forbidden on the organization network. For this reason, many organizations with high security requirements do not allow SSH to traverse and/or leave their network. Organizations that have less stringent requirements use this ability to secure their network. (You can also use SSH to create a VPN to carry all IP protocols, but that's in Chapter 13.) For example, I manage my website and blog with WordPress. It provides a friendly pointy-clicky interface for website administration and design, giving me a decent-looking page without me actually needing to learn *anything* about web design.[1] At one time, in those dark days before Let's Encrypt, my website used plain HTTP. I used SSH port forwarding to tunnel HTTP between my Web server and my desktop. This protected my credentials in transit and eliminated the risk of my password being stolen on the wire. This is a sensible and legitimate use of SSH port forwarding.

Suppose my desktop is inside a high-security network, however. The firewall tightly restricts web browsing and blocks all file transfers. If I can use SSH to connect to a server outside the network, I could forward my desktop's traffic to that outside server to get unrestricted Internet access. I could upload confidential documents over SSH, and the firewall logs would show only that I made an SSH connection. Your network administrator would object, with good reason.

Port Forwarding versus Security Policy

If you're an organization's security officer, port forwarding might make you consider entirely blocking SSH. I understand. I've had your job. You should also know that a recalcitrant user can tunnel SSH *inside* DNS, HTTP, or almost any other service or protocol, including raw ICMP. The only way to absolutely block SSH is to deny all TCP, UDP, or ICMP connections, use a web proxy that intelligently inspects traffic, and not allow your client machines access to public DNS even through a proxy. I've seen one firm actually implement this type of security perimeter, and they had many gaps and exceptions for notably clunky business-critical software. If you cannot implement this in your environment but have stringent security requirements, you must work with your users to meet those requirements and the business needs. I strongly recommend establishing a solid network traffic awareness program as well as intrusion and extrusion

detection, so you know when your network traffic deviates from the norm. Read Richard Bejtlich's books on intrusion and extrusion analysis, as well as my own *Network Flow Analysis* (No Starch Press, 2010), and implement programs like those discussed.

As a user, having the ability to tunnel arbitrary traffic over SSH does not mean you should do so. If your organization's security policy forbids port forwarding and/or tunneling, don't do it. If the policy says "use the web proxy and stay off IRC," then listen. I am not responsible if you use these techniques and are reprimanded, terminated, or exterminated. (Even if we IT security officers are all petty tinpot despots who don't understand your very personal and deeply urgent *need* for IRC and MySpace.)

Troubleshooting Port Forwarding

Some applications misbehave when used over port forwarding. It's important to separate application failures from port forwarding failures. If you've forwarded a port and your application doesn't work over it, use `netcat` or even `telnet` to determine if the port is actually open. (I demonstrated `netcat` at the beginning of Chapter 3.) The server should send the same feedback to a `netcat` request over a forwarded port as it does over a non—forwarded port.

If you don't get a response, you've probably misconfigured port forwarding. Double-check your command line. If necessary, use the debugging on one or both sides of the connection to see what's really happening. Remember that only one process may open a given port at a time.

If port forwarding works, your application has trouble with it. Perhaps you need a hosts entry, as is common with many web applications. Maybe it's an old and clunky protocol that expects a wide variety of ports open. FTP is a classic example. You'll need to dive into the application and its protocol to figure out why it's not working.

Port forwarding is a tool. Not all protocols work with this tool. Sometimes, using port forwarding is like trying to drive screws with a hammer; any result you get will displease you.

Example Environment

For all of these port forwarding examples, I assume that the SSH client is behind a firewall. This might be anything from a great big corporate proxy to a home router. There are several other servers behind this firewall, including web and email servers. The client is inaccessible to the public Internet; the outside world cannot connect to it.

The SSH server is on the public Internet. Anyone can connect to it, and it can freely access the rest of the Internet.

Port Forwarding Types

The three types of port forwarding are local, remote, and dynamic.

Local port forwarding redirects one port on the client to one port on the server. Essentially you're saying "Grab such-and-such port on the SSH server and make it local to my client." Suppose you want to download your email from a server that only offers unencrypted POP3, but you have SSH access to the server. You can forward, say, port 2110 on your local machine to port 110 on your POP3 server. Configure your email client to download its messages from port 2110 on the local host address. SSH intercepts all requests to port 2110 and patches them through to the mail server's port 110. Figure 9-1 illustrates the data flow of local port forwarding.

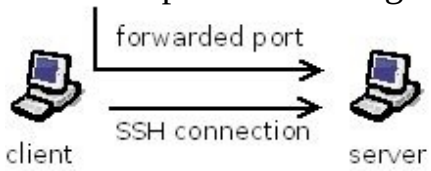


Figure 9-1: Local Port Forwarding Data Flow *Remote port forwarding* works in reverse. A port on the SSH server is forwarded to a port on your SSH client. You're saying, "take such-and-such port on my client and attach it to the remote server." For example, you could enable `sshd` on your workstation behind the corporate firewall. Then you SSH from your workstation to your server on the public Internet. With remote port forwarding, you could forward port 2222 on your public Internet server to the SSH port on your workstation. Anyone who connected to port 2222 on your public server would be transparently connected to your workstation's SSH server. They could get inside the firewall without any VPN client and with complete disregard for firewall policies. You might use remote port forwarding to make a private web server publicly available. Figure 9-2 illustrates remote port forwarding.

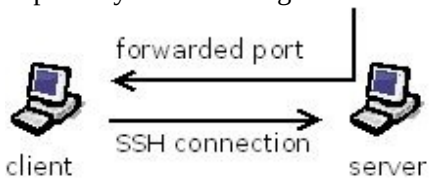


Figure 9-2: Remote Port Forwarding Data Flow *Dynamic port forwarding* is a broader system, where many different client programs can connect to many different services. It creates a SOCKS proxy on the SSH client, and tumbles any requests to that proxy out through the server. A SOCKS proxy is a generic gateway that can carry any TCP/IP traffic. (SOCKS doesn't actually stand for anything, by the way.) This gives anyone who connects to the proxy complete access to the server's network. Figure 9-3 illustrates dynamic port forwarding.

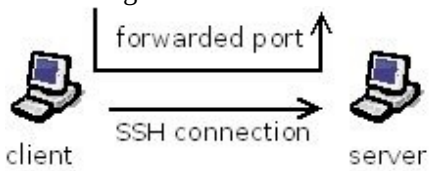


Figure 9-3: Dynamic Port Forwarding Data Flow When the underlying SSH session dies, all ports stop being forwarded. Chapter 10 offers suggestions for keeping SSH sessions alive.

With these possibilities, it's easy to see why sysadmins love SSH, and why many corporate security departments forbid it.

Privileged Ports and Forwarding

On Unix-like systems, TCP ports below 1024 are reserved for system use. Only `root` can bind to these ports. As an unprivileged user, you can attach the local end of your SSH port forwarder to any port above 1024. Forwarding a reserved port requires using SSH as `root`. Performing routine tasks as `root` is poor practice, so don't do it without a really good reason.

Only the side of the connection that is attaching to a privileged port needs to run as `root`. If you're binding a reserved port on the client, run the client as `root` but to log into the server as a regular user. If you're binding a reserved port on the server, you'll need to log into the server as `root`. In the latter case, it's a better idea to change the port so you don't have to login directly as `root`.

Microsoft systems do not implement privileged ports. Anyone can bind to any open port on the system. The absence of port restrictions creates all sorts of potentially amusing security issues, but it does make forwarding low-numbered reports no more difficult than forwarding any other port. You never need to run PuTTY as `root`.

Local Port Forwarding

Before setting up local port forwarding, verify that normal SSH works. Then figure out what service you want to forward, and what port that service runs on. Some typical choices are 80 (HTTP), 25 (SMTP), and 110 (POP3). The services that usually run on these ports are not normally encrypted.

Now choose a local port you want to use for the forwarding. Some clients work well when run on any port. Almost any mail client lets you set a TCP port to check POP 3 on. Others… don't. Websites frequently choke if you change the port number. If you don't know how the protocol behaves when forwarded from one port to another, try it on a test server and see.

For our local port forwarding examples, we'll forward port 8080 on my client to port 80 on the server `sloth`. Now that TLS certificates are free, why would you need to do this? Some proprietary web-based applications don't support TLS, and if you try to convert them to TLS they die screaming.[2] I'll need to edit the client's hosts file (either `/etc/hosts` or `C:\Windows\System32\drivers\etc\hosts`) to tell my client that the website has the IP address 127.0.0.1. I'll need a second alias so that I can SSH out to the actual machine. If I'm the only one that uses this application, once I have port forwarding setup I could tell the application to only listen on the server's local host address. This would not only protect my data in transit as TLS would, it would add another layer of protection for the application.

OpenSSH Local Forwarding

To tell the SSH client to activate local forwarding, use the `-L` flag.

```
$ ssh -L localIP:localport:remoteIP:remoteport hostname
```

If you don’t specify an IP address on the SSH client, SSH attaches to 127.0.0.1. You can skip the first argument in this case, making the command: $ **ssh -L localport:remoteIP:remoteport hostname**

For now, only use the IP address 127.0.0.1. This is the loopback address on every machine, accessible only on that machine. While it might look like we’re forwarding an address to the same address, 127.0.0.1 on the client is not the same as 127.0.0.1 on the server. We’ll consider binding a forwarded port to a different IP address in “Choosing IP Addresses” later this chapter.

So here’s how we use local port forwarding to connect to the server **`sloth`**, and forward port 80 on the localhost address of **`sloth`** to my client’s port 8080.

```
$ ssh -L 8080:127.0.0.1:80 sloth
```

I’m attaching to port 8080 on my workstation. I haven’t specified a local IP address, so `ssh` attaches the forwarding to the client’s 127.0.0.1. My SSH session logs on normally, and gives me a terminal on the server. But if I point my web browser to localhost:8080, I’ll be connected to the website running on the server. An alias in the hosts file will make the website much more usable.

To set up local port forwarding every time you connect to a server, use the LocalForward keyword in *`ssh_config`*.

```
LocalForward client-IP:client-port server-IP:server-port This looks like port forwarding
on the command line, but the middle colon is missing. Here I forward port 8080 on my
workstation to port 80 on the server. We attach to the 127.0.0.1, or localhost, on both
the client and the server. I’m using port 8080 on the workstation because using port 80
would require running SSH as root every single time.
Host envy.mwl.io
LocalForward localhost:8080 localhost:80
```

The LocalForward keyword most often appears with a Host statement, enabling local port forwarding when you connect to specific servers. To avoid IP and port conflicts, each server usually gets assigned its own local port.

PuTTY Local Forwarding

PuTTY has a special control panel just for port forwarding. On the PuTTY Configuration screen’s left side, select *Connection -> SSH -> Tunnels*, as shown in Figure 9-4.

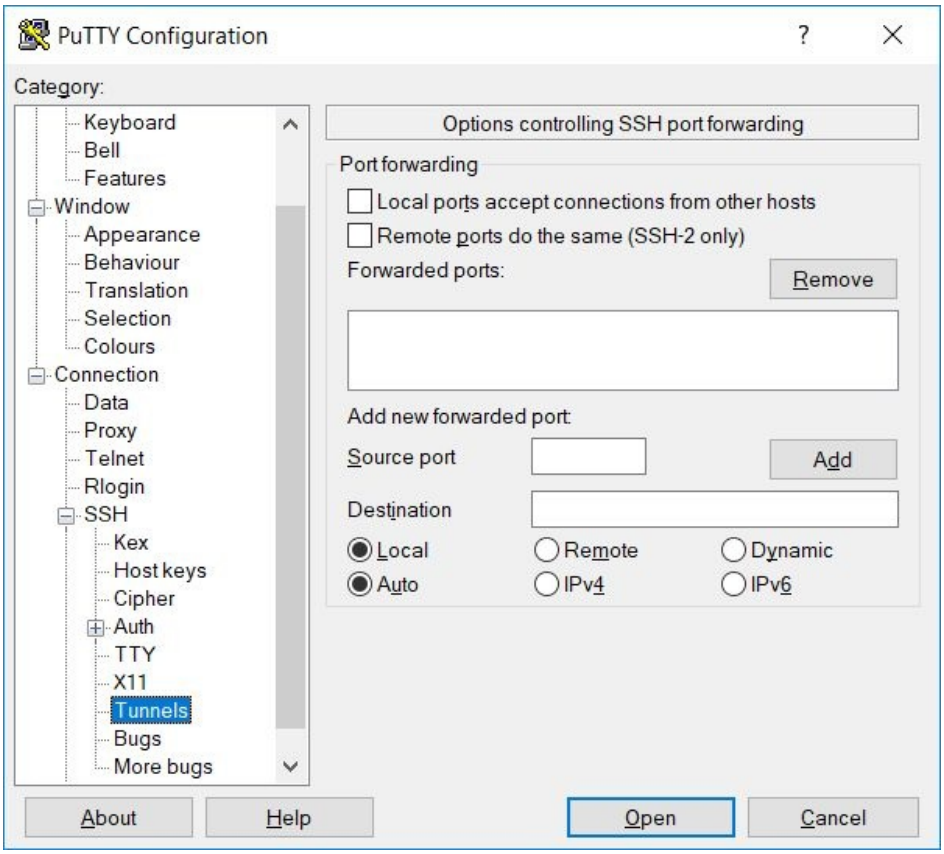


Figure 9-4: PuTTY Port Forwarding With local port forwarding, PuTTY attaches to the client's localhost address by default. I must specify the address on the SSH server to use, however. To forward port 80 on the SSH server to port 80 on my workstation, I'll use the server's localhost address. In *Source port*, enter 80. In *Destination*, enter the IP address on the server, a colon, and the port to be forwarded. Here I'll use 127.0.0.1:80. At the bottom, select *Local*. It should look like Figure 9-5.

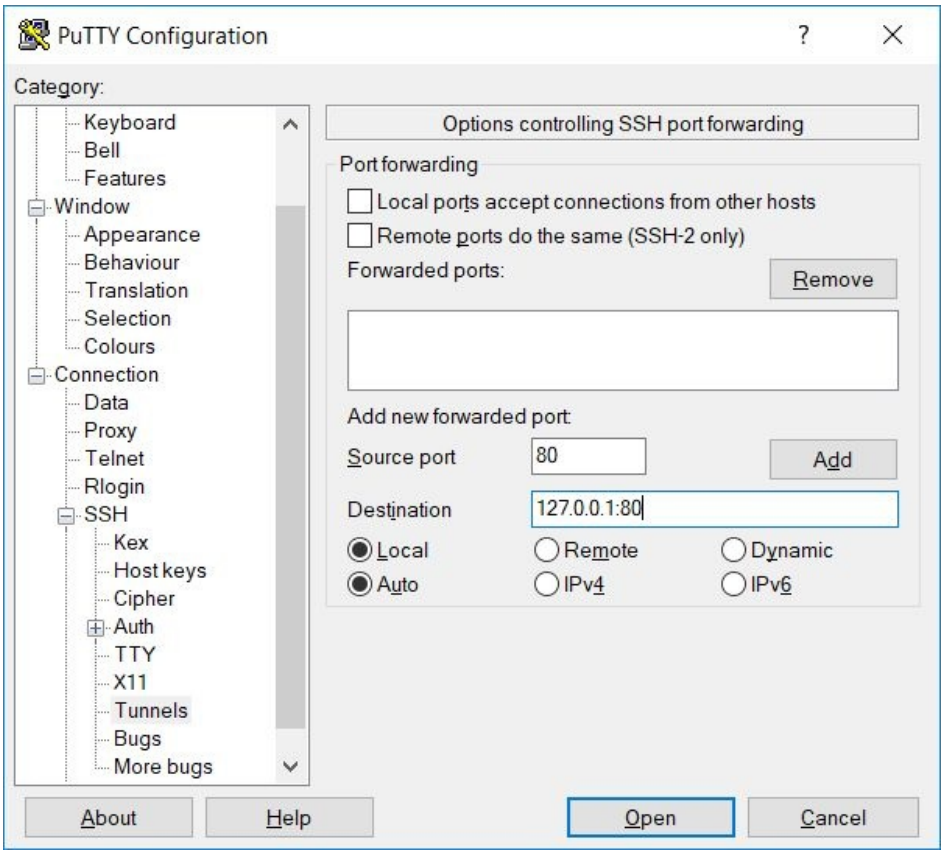


Figure 9-5: PuTTY Local Port Forwarding Settings Hit *Add*, then connect. You now have port forwarding. Point your browser at **`localhost`**, and see what happens.

To bind this forwarding to the client's network-facing IP address, select *Local ports accept connections from other hosts*. This binds the forwarded port to all IP addresses on the client, so that other hosts on the workstation's network can access the forwarding. See "Choosing IP Addresses" later this chapter for a discussion of the implications.

If you want to use this forwarding every time you connect to this host, save this session.

Remote Port Forwarding

Before configuring remote port forwarding, verify that normal SSH works. Determine the client and server ports you want to forward to and from.

Where local port forwarding is usually used to wrap a service with encryption, remote port forwarding is used to access a service behind a firewall. For this example, I'm going to forward port 2222 on the SSH server's localhost address to port 22 on the workstation. When I connect to port 2222 on the SSH server, remote forwarding will redirect me to the workstation's SSH service.

Why do this? Remember from our example environment, the client is behind a firewall. The firewall might be my home NAT device, or my employer's industrial-grade corporate firewall cluster. Remote forwarding lets me use my

client to give an SSH server outside the network a way to connect to a host inside the firewall, despite any firewall rules to the contrary. This might be my invaluable emergency back door into my own network, or it might violate my employer's security policy. Or, better still: both!

Note that you cannot bind a forwarded port to the SSH server's public-facing IP addresses unless the server is specifically configured to permit this with the GatewayPorts keyword. See "Restricting Port Forwarding" later this chapter.

OpenSSH Remote Forwarding

Configure remote port forwarding with the `-R` flag.

```
$ ssh -R remoteIP:remoteport:localIP:localport hostname
```

If you don't specify an IP address to attach to on the SSH server, SSH attaches to 127.0.0.1. You can skip the first argument in this case, making the command: $ **ssh -R remoteport:localIP:localport hostname**

I want to connect port 2222 on the SSH server **`sloth`** to port 22 on my workstation, using the localhost address on both sides.

```
$ ssh -R 2222:localhost:22 sloth
```

My client connects to the server and gives me a command prompt. As long as that SSH session remains open, another user on **`sloth`** could SSH to my workstation by connecting to port 2222.

```
sloth$ ssh -p 2222 localhost
```

Poof! A new SSH connection into my workstation, tunneled inside my existing SSH session. This new session would show up in the client log as a new connection, originating from the localhost. You really need to trust the people who have accounts on your systems when setting up remote port forwarding. Anyone who can access your system's localhost address can use the port forwarding. I would never use remote port forwarding on an SSH server I didn't wholly trust.

If you want to establish remote port forwarding every time you connect to a server, use the RemoteForward keyword in *`ssh_config`*.

```
RemoteForward server-IP:server-port client-IP:client-port Once again, this resembles port
forwarding on the command line, but the middle colon is missing. Here I set up this same
port forwarding in the configuration file.
Host sloth.mwl.io
RemoteForward localhost:2222 localhost:22
```

The RemoteForward keyword most commonly appears with a Host statement, unless you want to perform remote forwarding on every host you connect to.

PuTTY Remote Forwarding

To configure remote forwarding, go to the PuTTY Configuration screen's left side, select *Connection -> SSH -> Tunnels*, as Figure 9-4 shows. As we're forwarding from server to client, the *Source port* field refers to the port on the server that will be forwarded to the workstation. In this case, the source port is

2222. The *Destination* is localhost:22, because the workstation's SSH server runs on port 22.[3] Select *Remote* for remote port forwarding.

Hit *Add*, then connect. Port forwarding should work.

To bind this forwarding to the server's network-facing IP address, select *Remote ports do the same (SSH-2 only)*. This binds the forwarded port to all IP addresses on the SSH server, so other hosts can access the forwarding. "Choosing IP Addresses" later this chapter discusses the implications.

To make the remote forwarding permanent for this server, save the session.

You can now laugh at the firewall all the way to the unemployment office. Or get into your network when the VPN fails, saving your company. Or, again, both.

Dynamic Port Forwarding

Dynamic port forwarding transforms your SSH client into a SOCKS (version 5) proxy. Any traffic sent to the proxy will be tunneled to the SSH server, which forwards that traffic as its own access permits. You must have a SOCKS-aware application to access the proxy, but most web browsers include SOCKS support. In this example, I'm going to configure port 9999 on my workstation as a SOCKS proxy and dynamically forward all traffic to my server on the public Internet.

When using SOCKS, your client will probably need to forward all DNS requests to the SOCKS server. Not all clients support this.

OpenSSH Dynamic Forwarding

Use the `-D` flag to tell OpenSSH to use dynamic port forwarding.

```
$ ssh -D localaddress:localport hostname
```

If you don't specify an IP address, `ssh` automatically binds to 127.0.0.1.

Here, I create my proxy on port 9999 on my workstation. All traffic sent to the proxy gets forwarded to the SSH server **`sloth`**, which relays it to its destination.

```
$ ssh -D 9999 sloth
```

As usual with port forwarding, you'll log on to the server and get a command prompt. The dynamic forwarding runs in the background. Configure the web browser on the workstation to use the SOCKS proxy at 127.0.0.1:9999. It should send all your browsing over the SSH connection to your server.

If you want remote port forwarding configured every time you connect to a host, use the DynamicForward keyword in *`ssh_config`*.

```
DynamicForward host:port
```

Like the other forwarding statements, and for the same reasons, the DynamicForward keyword most commonly appears in a Host statement.

PuTTY Dynamic Forwarding

Go to the *Tunnels* screen shown in Figure 9-4. In the *Source port* field, enter the

port that you want your SOCKS proxy to use. Leave *Destination* blank. Select *Dynamic*, then hit *Add*. You'll see the port forwarding appear in the *Forwarded ports* list. Open the connection. Your browser should now be able to connect via the SOCKS proxy.

For my sample use, I enter 9999 in the *Source port* field, select *Dynamic*, hit *Add*, and connect. That's it.

To bind this forwarding to the client's network-facing IP address, select *Local ports accept connections from other hosts*. This binds the proxy to all IP addresses on the workstation, so other hosts can access the forwarding. Remember that you're offering the tunnel to everyone who can access your client when you do this.

Save the session if you want this forwarding started automatically every time you open this connection.

Testing Dynamic Forwarding

You can verify dynamic forwarding with any program that supports SOCKS proxies. The most common program of this type is a web browser.

Configure your firewall to block all port 80 traffic from your workstation. Verify that you can no longer browse the web. If you're going to browse, you'll need to do it over proxy.

Start a dynamic port forwarding SSH session. Configure the web browser to access that proxy. If you can see the Internet, dynamic forwarding is working.

Backgrounding OpenSSH Forwarding

Sometimes you want to use OpenSSH to forward a connection, but you don't need a terminal session on the SSH server. Use the `-N` flag to tell `ssh` to not run anything, including a terminal, on the server, and the `-f` flag to tell `ssh` to go into the background on the client. Here I background a local forwarding session to the server **`pride`**.

```
$ ssh -fNL 2222:localhost:22 pride &
```

Backgrounding this command gives you your original terminal back. Backgrounded forwarding is useful when you do not have shell access on the SSH server, but you are allowed to authenticate yourself and create a tunnel. (This is one way to create an SSH-based VPN, but Chapter 13 discusses better ways.)

Choosing IP Addresses

When port forwarding, you must choose the IP address you want the forwarded port to listen on, and the IP you want to attach the forwarded port to. Choosing the IP helps control who may connect to the forwarded port.

The most common choice is to bind to the localhost address, 127.0.0.1, on

either or both ends of the tunnel. Every machine with a functional TCP/IP stack uses 127.0.0.1 as the address for itself, and only the local machine can connect to it. If I forward port 80 on my workstation's localhost address to port 80 on the server's localhost address, no other hosts can connect to that forwarded port over my tunnel. Most daemons on a server listen to the localhost address as well as one or more network-facing IP addresses, so using the localhost address is a reasonable way to forward ports.

If you want your client to accept requests from other machines and use local port forwarding to send them to the SSH server, attach the port forwarding to the client's network-facing IP address. If I forward port 80 on my machine's network-facing IP address to port 80 on the SSH server, this forwarding is available to all hosts that can connect to my client's port 80. With PuTTY, you must select *Local ports accept connections from other hosts*. With OpenSSH, you must have a GatewayPorts keyword set in `ssh_config` (see "Gateway Ports" later this chapter.) If you want the SSH server to forward requests from other machines to your client using remote port forwarding, attach the port forwarding to the server's network-facing IP address. You must adjust GatewayPorts in `sshd_config` as shown in "Gateway Ports" later this chapter. For example, we used remote report forwarding to connect a port on our server to the client's `sshd`. You could attach this remote forwarding to the server's public facing IP address, so that any host on the Internet could connect to the client's SSH service even though it's behind a firewall. Remember, while creating a back channel into a private network might be useful, opening that back channel to the entire Internet is downright gauche.

If you want an SSH client to act as a SOCKS proxy for other machines via dynamic port forwarding, attach the port forwarding to the client's network-facing IP address.

Always remember that a host running any modern OS can have multiple IP addresses. It might make sense for you to pick a particular address rather than allowing all network-facing addresses.

Suppose my workstation has an IP of 192.0.2.18 and is on a network with a whole bunch of other clients. We have to access a critical web-based application that doesn't encrypt data in transit. I can provide an encrypted tunnel from my workstation to the server via local port forwarding. If I wanted to provide this tunnel to my desktop alone, I would attach the client's end of the tunnel to 127.0.0.1. If I wanted to offer this tunnel to everyone on my network, I would attach the client end to 192.0.2.18.

Or maybe I'm responsible for running the company's content-filtering web proxy and I'm trying to debug a problem where a certain website doesn't

function through the proxy. I want to see what this website looks like from outside my network. I could set up a private SOCKS proxy to bypass the organization's proxy, letting me browse from the outside server instead. Setting up an unauthorized proxy server that anyone can use is a great way to need a new job, so I make absolutely sure that the local end of that tunnel uses the localhost address.

You can use a hostname instead of the actual IP address, provided that the hostname appears correctly in the DNS. You can also use the word `localhost` instead of 127.0.0.1.

Restricting and Requiring Port Forwarding

The OpenSSH server controls what types of port forwarding users can perform. You can either deny port forwarding, permit port forwarding but allow binding only to the localhost address, or permit only specific addresses and ports.

Implementing these blocks at the server level isn't as effective as one might hope, though. A user who has shell access can easily install their own forwarders. Properly disabling forwarding for shell users requires controlling which binaries are executable, disabling interpreters like Perl or Python, and preventing users from installing further programs. For the most part, unless you're *really* dedicated, users with shell access can figure out ways to forward ports. Still, disabling or restricting port forwarding will give your users a really solid hint that they shouldn't be forwarding ports.

Block Port Forwarding

The *`sshd_config`* keyword AllowTcpForwarding tells `sshd` whether it should permit port forwarding. The default is `yes`, allowing port forwarding. If set to `no`, port forwarding is completely disallowed.

To permit only local port forwarding, set AllowTcpForwarding to `local`. Similarly, `remote` permits only remote port forwarding.

Gateway Ports

The GatewayPorts keyword controls whether a client can bind a remote forwarded port to any IP address other than `localhost`. This keyword appears in both *`ssh_config`* and *`sshd_config`*. The *`ssh_config`* option controls local port forwarding, while the *`sshd_config`* option controls remote port forwarding.

GatewayPorts is set to `no` by default, meaning that clients cannot connect any port forwarding to any network-facing IP address. This is identical in both *`ssh_config`* and *`sshd_config`*.

When used in *`ssh_config`*, setting GatewayPorts to `yes` to allows `ssh` to request to listen to any IP on the client.

On the server side, setting GatewayPorts to `yes` in *`sshd_config`* means that no

matter what the client requests, remote forwarding always listens to all addresses on the host. I have no idea why you'd enable global network access on all port forwardings, but it's an option.

The server supports one additional GatewayPorts option in *sshd_config*, `clientspecified`, which tells `sshd` to let a client bind to whatever they request. Permitting the client fine-grained control on a forwarding-by-forwarding basis is usually the best choice.

Allow Specific Ports and Addresses

If you want more specificity than GatewayPorts supports, you can restrict which TCP ports and addresses can be forwarded with the PermitOpen keyword in *sshd_config*. PermitOpen takes a space-delimited list of ports that may be forwarded in the form of `hostname:port`. For example, here I permit the server's ports 25 and 110 to be forwarded back to the client, and only from the localhost address.

```
PermitOpen localhost:25 localhost:110
```

Anything not permitted is forbidden. The SSH session will open normally, but when you attempt to pass traffic over a forbidden forwarded port your SSH client displays an error.

Requiring Port Forwarding

Perhaps the port forwarding is the only reason for this connection to exist. If setting up port forwarding fails, you don't even want the connection to establish. The ExitOnForwardFailure *ssh_config* keyword tells `ssh` what to do in the event an attempt to forward a port fails. The default, `no`, means the connection should be set up even if port forwarding cannot be established. By setting ExitOnForwardFailure to `yes`, you tell SSH to immediately disconnect if the port forwarding doesn't work.

Now that you know how to selectively forward ports to help glue your network together, let's see how to keep an SSH session alive for hours or days at a time, without human intervention.

---

[1] My HTML education ended about 1996, and I have no desire to resume it.

[2] Yes, we're solving the wrong problem here. The real fix is to replace the boneheaded application.

[3] I know, I know, most Windows systems don't have an SSH service. I'm choosing to keep my examples consistent, rather than confuse you further.

Chapter 10: Keeping SSH Connections Open

Port forwarding transforms SSH from a protocol that gets you a terminal session into a tool for arbitrarily forwarding TCP traffic. But most firewalls (and some Internet service providers) deliberately terminate TCP connections left idle for a period of time. SSH sessions left idle will eventually be disconnected by the server, the client, or some network device in between. If you're forwarding a service over SSH, or even if you're too lazy to log into your SSH server every time the firewall cuts your connection, you want to keep your session alive.

Most methods for keeping an SSH connection up amount to "pass a small amount of traffic in the background so that intermediate network devices don't see the connection as idle." These are called *keepalives*. Running a program that continuously displays and updates, like top(1), can act as a keepalive without changing any SSH settings. All you need to do is get in the habit of starting `top` every time you're interrupted.

The problem with keepalives is that temporary disconnections terminate the session. If your service provider has a problem in the middle of the night and the keepalive packets cannot cross the network for a few minutes, either your client or your server will terminate the connection. Decide how to configure keepalives appropriately for your network. You might not want them at all.

If your connection is so erratic that keepalives can't sustain your connection, investigate mosh (https://mosh.org). It's a remote connection protocol similar to SSH, but designed for unreliable networks.

You have two options for keepalives, TCP keepalives and SSH keepalives.

TCP Keepalives

Both PuTTY and OpenSSH support TCP keepalives. While TCP keepalives are not as configurable as SSH keepalives, they're sufficient for most end-users.

A TCP keepalive is part of the TCP protocol, is sent at the transport layer, and is not part of SSH itself. When a TCP connection remains idle, it eventually times out and disconnects. Turning on TCP keepalives sends occasional packets back and forth just to remind everyone that this connection is still here. A TCP keepalive can be spoofed or forged, though. This is not necessarily bad—I can't imagine why anyone would want to spoof your connection to keep it alive, but someone more clever and more nasty than I can probably come up with more than one bad reason. How often you need to send a TCP keepalive depends on your operating system's TCP stack, but it should never be longer than two minutes.

PuTTY only supports TCP keepalives, but doesn't originate them by default.

It responds to any TCP keepalives it receives, however. On the PuTTY Configuration screen, go to the Connection section. The first option is *Seconds between keepalives*. This defaults to zero, disabling sending keepalives. In most cases, sending a TCP keepalive every 90 seconds suffices to hold the connection open. Even if PuTTY doesn't send keepalives, SSH servers usually do, and PuTTY responds to them. This usually suffices to hold the connection open.

The OpenSSH server sends TCP keepalives by default. If you want to disable them, set the keyword TCPKeepAlive to `no` in *`sshd_config`*.

OpenSSH Keepalives

While TCP keepalives might meet most people's needs, OpenSSH's keepalives are much more flexible. The keepalive messages, sent within the encrypted channel, tell intermediary network devices that this TCP session is still in use. Receiving a keepalive tells the host that the remote end is still connected, and that the SSH session is still valid. An SSH keepalive is also more likely to continue holding a session open even through a lengthy router reboot.

Both OpenSSH's client and the server support keepalives. Strictly speaking, the client sends *client alive* messages and the server sends *server alive* messages. While these must be different for protocol reasons, to us they're both just keepalives. OpenSSH doesn't use SSH keepalives by default; you must configure them before starting a session.

A host that sends keepalives expects to receive keepalives in return. Each host tracks how long it's been since it received a keepalive from the other end. If a host sends a specified number of keepalives without receiving any, it assumes that the connection is lost and terminates the SSH session.

Using SSH keepalives requires deciding how often you want to send a keepalive packet, and how many of those packets can be missed before the host disconnects the session. The server uses the keywords ClientAliveInterval and ClientAliveCountMax. The client supports the keywords ServerAliveInterval and ServerAliveCountMax.

The AliveInterval keywords dictate how many seconds the connection must be idle before the host sends a keepalive. To make a client transmit a keepalive after ninety seconds of inactivity, set ServerAliveInterval to 90. The default is 0, disabling keepalives.

The AliveCountMax keywords tell the host how many keepalives it must send in a row before terminating the connection. The default is three.

Let's look at how this works in practice. We have the following in the server's *`sshd_config`*: ClientAliveInterval 90

```
ClientAliveCountMax 5
```

On the client side, we've put the following in *`ssh_config`*.

```
ServerAliveInterval 90
ServerAliveCountMax 4
```

We log into our SSH server, do some work, and let the connection go idle. Ninety seconds after the connection goes idle, the client sends a keepalive to the server. If the server responds with its own keepalive, both client and server know that the connection is alive. If another ninety seconds pass without receiving a response from the server, the client will send another keepalive. It knows that it's sent two keepalive requests without receiving any response from the server. If the connection remains idle, the client keeps sending keepalives. At the fourth keepalive, after six minutes, the client throws away the SSH session and exits.

The server sends keepalives in the same way, but note that it's set to tear down the connection at five unacknowledged keepalive requests. This particular client tolerates less interruption than the server.

Note that the TCP protocol also plays into this. A host sending TCP packets expects the recipient to acknowledge every packet. If the sender does not get this acknowledgment, it eventually tears down the connection despite anything SSH can do. The length of time varies by operating system, but you should know that if you cannot maintain a TCP connection you cannot maintain an SSH session.

If you want to keep your connection alive no matter what, cranking AliveCountMax to high values helps, especially when you're behind a cheap[1] Network Address Translation device such as many home routers.

PuTTY does not support SSH keepalives.

Keepalives and the SSH Server

If you disable all keepalives on your SSH server, the server cannot notice when a client goes off-line. This means that when a workstation crashes or a network link fails, forcibly disconnecting a client, the server won't know. It will continue running the SSH processes for these clients. If your server is up for a long time, you may accumulate hundreds or even thousands of defunct `sshd` processes. Cleaning them up is kind of a pain. I recommend using TCP keepalives at a minimum, and preferably SSH keepalives as well.

Now let's look at simplifying your life through key distribution.

---

[1] I'm fine with inexpensive, but I *detest* cheap.

Chapter 11: Key Distribution

Unquestionably, the most annoying part of managing SSH is distributing and verifying keys.

No matter how dire the lecture you inflict upon your users, many of them won't bother to compare server fingerprints to the list you provide; instead they'll hit "Yes, accept the key." No matter how hard we try to educate them, users quickly grow inured to the scary-looking warnings and learn to ignore them. The best way to help users pay attention is to ensure that they don't see warnings unless something is truly wrong.

Similarly, key-based authentication is usually more secure than password-based. Many users won't bother to copy their `authorized_keys` to a server, however. They'll just stick with familiar passwords. If you want to enforce key-based authentication, you'll need to get the user's `authorized_keys` on the servers yourself. And if you manage dozens or hundreds of servers and/or users, you will need automation to distribute user key updates amongst your systems.

While OpenSSH doesn't include automated key distribution tools, understanding key-related features can vastly simplify your automation process. We'll start with host keys, and proceed to user keys.

known_hosts In Detail

Host key distribution, for both OpenSSH and PuTTY, starts with `known_hosts`. If you're going to distribute host public keys, you'll want to be sure that those records are pristine. That means you need to completely understand the `known_hosts` file.

Each line in `known_hosts` represents one public key from one host, in space-separated fields. If a host supports three different public key algorithms, and you've connected to this host using all three keys, that host will have three entries in `known_hosts`. Each entry also gives the server's hostname or IP address and the algorithm used for the key. But each entry can also include a couple other fields.

Marker

The `known_hosts` file supports to special markers, @cert-authority and @revoked. These markers must appear first in line.

A `known_hosts` entry that starts with *@cert-authority* indicates that the host key is for an SSH certification authority. An SSH certification authority is not the same as a TLS CA. Chapter 14 discusses SSH CAs.

If an intruder breaks into an SSH server and copies the servers private key, that key can no longer be trusted. A savvy intruder might use that key to try to

spoof the server. By marking a key with @revoked in *known_hosts* you tell ssh to not accept this key and to generate a scary warning.

```
$ ssh gluttony
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@ WARNING: REVOKED HOST KEY DETECTED! @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
The ECDSA host key for gluttony.mwl.io is marked as revoked.
This could mean that a stolen key is being used to impersonate this host.
ECDSA host key for gluttony.mwl.io was revoked and you have requested strict checking.
Host key verification failed.
```

Note that there is no “accept this key anyway” option. A revoked key is utterly un-trusted. Leaving the key in *known_hosts* but marking it as a revoked gives the user clear warning that they’ve encountered a compromised system.

Markers must go at the beginning of the line, before the hostname.

Hostname

The hostname is how SSH identifies an SSH server. If you used a short hostname to connect to the server, ssh records the full hostname that it used to contact the server. This means that if I typed `ssh wrath`, ssh would record the hostname as **wrath.mwl.io** because that’s the name my system’s resolver provided to ssh(1). The machine might have other host names or aliases, and is probably also known by its IP address. A truly authoritative *known_hosts* file must include keys for each of those names.

The good news is, you don’t have to include multiple mostly-duplicate lines for these different names. The *known_hosts* file accepts multiple host names in a single entry, so long as they are separated by commas.

```
gluttony.mwl.io,mail.mwl.io,203.0.113.213 ecdsa-sha2-nistp256
AAAAE2VjZHNhLXNoYTItbmlzdHAyN…
```

Some sysadmins change the TCP port their SSH service runs on. This isn’t terribly useful for security, but helps slow down the more primitive worms and reduces log chatter. These host names appear in brackets in *known_hosts*, followed by a colon and the port number.

```
[lust.mwl.io]:2222 ecdsa-sha2-nistp256 AAAAE2VjZHNhLXNoYTItbmlzdHAyNTYAAAAIbmlzdHAyNT…
```

Chapter 5 covered obscuring host names by hashing them, preventing a casual intruder from extracting server information from *known_hosts*. Listing multiple host names on a single line simplifies central management of *known_hosts*, but conflicts with hashing host names. If you wish to hash host names, you must list each hostname on a separate line. A host known as **avarice.mwl.io**, **mail.mwl.io**, and **198.51.100.12** requires three *known_hosts* entries, and each will be separately hashed. If you’re going to hash *known_hosts* entries before distributing them, I recommend maintaining your master file in clear text.

A host that accepts connections on multiple IP addresses theoretically needs a *known_hosts* entry for each of those addresses. If you don’t normally connect to all of those addresses, then don’t bother. I have a server with dozens of IP

addresses, but I only connect via SSH to one of those addresses, so that server has only one *known_hosts* IP entry. If you have such a server, locking sshd to only listening on one address might simplify management.

Key Type

The key type is the algorithm used to generate this host key. A modern *known_hosts* can contain six different key types: *ssh-dss* (DSA keys), *ssh-rsa* (RSA keys), *ecdsa-sha2-nistp256*, *ecdsa-sha2-nistp384*, and *ecdsa-sha2-nistp512* (ECDSA keys), and *ssh-ed25519* (ED 25519 keys). Anything else that appears in the space is weird and needs investigation.

Key

The public key is a long gibberishy alphanumeric string. It often starts with a series of capital A's and often (but not always) ends with equal signs (=). The key fills the majority of the line.

Comment

The comment is free-form text. You can use the comment anyway you need. It's generally blank in automatically-maintained *known_hosts* files, but you'll find it useful in centralized management.

Creating known_hosts

The easy way to generate a *known_hosts* is to use ssh-keyscan(1).

```
$ ssh-keyscan wrath > wrath.known_hosts
```

That gives you a *known_hosts* file to start with. Now you need to verify those keys against the fingerprints you generated in Chapter 4. That's a great job to give to a meticulous, conscientious flunky you loathe.

I encourage you to automate collecting *known_hosts* entries. How you do this depends entirely on your organization's preferred tools. Ideally, you'd run ssh-keyscan when the machine is first deployed, before any intruder has a chance to trash it, and immediately update your *known_hosts*.

If you want to simplify *known_hosts*, you could reduce the number of keys that an SSH server offers. You might declare that all hosts in your network only offer ED25519 keys, eliminating all the *known_hosts* entries for all other key types. The *sshd_config* HostKeyAlgorithms keyword lets you set the algorithms sshd will use for host identification.

```
HostKeyAlgorithms ssh-ed25519, ssh-rsa The exact method you'll use depends entirely on the
tools you're comfortable with and the automation you already have in place. If at all
possible, repurpose your existing tools.
```

And if you write a good tool to collect, verify, and build a *known_hosts* file, please make it publicly available.

Revoking Host Keys

If you have reason to suspect that a server's key has been compromised, revoke it. Find all of the server's host key entries in your *known_hosts*. Add to the string

@revoked in front of all of them. Generate new host keys for the server and restart `sshd`, then add the new host keys to your *`known_hosts`*. You can now distribute your updated *`known_hosts`* to your clients, and in the (unlikely) event that the user attempts to use the revoked key, the user will get a warning.

The effectiveness of revoked keys depends entirely on distributing *`known_hosts`* to your clients.

Distributing Host Keys

Any time an SSH server's host key is added, moved, or changed, users will see warnings about the host key. The whole point of distributing *`known_hosts`* is to keep users from seeing unnecessary warnings. Stay ahead of your users.[1] Update your *`known_hosts`* any time you deploy or remove a server, or if you must give a server new keys. If you delay updating *`known_hosts`*, users will learn to ignore warnings.

The worst part of maintaining a centralized *`known_hosts`* file is copying the file to all of your servers and workstations. You're busy. If the update takes a long time or a lot of energy, you won't keep up on it. You really need a centralized system like Ansible, Puppet, or one of their many competitors. Active Directory works fine for distributing host keys to Windows systems. If you've never used automation, I recommend Ansible. Once you have a complete *`known_hosts`* for your existing systems, updating that file and pushing it out to all of your systems should only take a minute or two, and will save your users and your support team hours of labor.

Distributing known_hosts

All OpenSSH clients check */etc/ssh/ssh_known_hosts* for host keys. Copy your *`known_hosts`* to this location on each of your servers and workstations. The next time someone uses ssh(1) on these machines, the correct key will already be in place.

OpenSSH checks for host keys in each user's personal *`known_hosts`* file in addition to the system's */etc/ssh/ssh_known_hosts*. The client will use any entry that matches the key offered by the server. When you first deploy a centralized *`known_hosts`*, each user will probably have an existing personal *`known_hosts`*. You don't want any obsolete or invalid entries in the user's personal cache to interfere with later key changes or revocations. Don't just go deleting everyone's *`known_hosts`*; they might contain verified host keys for servers you don't control. Instead, on your first deployment, move each user's personal *`known_hosts`* to somewhere like *`known_hosts.personal`*.

Be sure to tell your users what's going on. Preferably in advance.

Once you have a system in place to maintain *`known_hosts`*, you'll find other uses for automation in SSH. Remember that */etc/ssh/ssh_config* sets systemwide

defaults for ssh(1). If you have organizational standards that require special settings, you can enter them in the global configuration and save your users the effort of editing their own configurations or remembering command-line arguments. If your organization runs SSH on a non-standard port, setting the Port keyword in `/etc/ssh/ssh_config` might actually earn you good karma from your users. Personal config files override system-wide settings, so users can still shoot themselves in the foot if they're really intent on it.

Distributing PuTTY Host Keys

PuTTY keeps its host keys in the Windows Registry. Copying the keys isn't as easy as moving a file to all of your workstations, but it can be simplified. The PuTTY team has a Python script to convert `known_hosts` into PuTTY's Registry keys, `hk2reg.py`. You won't find `hk2reg.py` in the normal PuTTY installation, but it's included with the source code. You can download the PuTTY source code from the PuTTY website, or grab it from the PuTTY GitHub at https://github.com/github/putty, in the "contrib" directory.

Run `hk2reg.py` and give it a single argument, your pristine `known_hosts`.

```
$ kh2reg.py pristine-known_hosts > putty.reg
```

Install this registry file on your clients via Active Directory, a login script, or by having your users double-click on it.[2]

Remember that PuTTY stores keys in each individual user's Registry. There is no systemwide PuTTY registry tree. Distribute keys by user, not by machine.

If you are maintaining `known_hosts` for a variety of platforms, I suggest this workflow for distributing host keys: Start by gathering your host keys. Create a `known_hosts` file for your OpenSSH clients. Trigger the script to automatically distribute the new `known_hosts` to each of your OpenSSH systems. While that runs, use `kh2reg.py` to create your Windows registry. Last, queue your new registry file for distribution via Active Directory. The next time people login, they should have all the new keys.

Host Keys in DNS

OpenSSH supports checking for host key fingerprints in the Domain Name System. (PuTTY does not.) This eliminates pushing the file to your servers, but traditional DNS services are not secure. You absolutely must have DNS Security Extensions (DNSSEC) if you want to securely distribute your servers public key fingerprints via DNS. If you do not yet have DNSSEC, go configure it now and then come back here. You might find my book *DNSSEC Mastery* (Tilted Windmill Press, 2013) useful.

We're not going to cover DNS basics. If you're considering distributing key fingerprints via DNS I'll take it as given that you know what a zone file is, why an RR is important, and why you update serial numbers.

SSHFP Records

The SSH Finger Print (SSHFP) record provides a host's SSH fingerprint. The record looks something like this: wrath IN SSHFP 1 1 07988cadf134050d458dfa5f2c062b5e68106163

As with any standard DNS record, the first field gives the hostname, the second indicates this is an Internet record, while the third indicates this is an SSH fingerprint record. SSH-specific details start appearing in field four, which gives the algorithm type. You don't have to memorize which number maps to which algorithms, but the 1 here means this is an RSA fingerprint. The fifth field is the message digest algorithm used to produce this fingerprint. 1 indicates SHA-1, while a 2 represents SHA-256. Finally, the sixth field is the actual key fingerprint.

You'll need two SSHFP records for every public key your server offers; one for SHA-1 and and one for SHA-256.

Creating SSHFP Records

Don't even try to create SSHFP records by hand. The `ssh-keygen` program can read the key files on the local server and produce records, by using the `-r` flag. Give the hostname as an argument.

```
$ ssh-keygen -r wrath
wrath IN SSHFP 1 1 07988cadf134050d458dfa5f2c062b5e68106163
wrath IN SSHFP 1 2 b7931f47398ca1ed73e8642bd029fb69dda05913058ffb096f2358c429436013
wrath IN SSHFP 2 1 3f73194323def663866a7b3996e6be113d7ea303
wrath IN SSHFP 2 2 927b54096876789ca926da1aa80db5a09751c8d9c5c99527b3a231e878802e3e wrath
IN SSHFP 3 1 cf61d5ed8a653750198daf77f0a409d48c8ef760
wrath IN SSHFP 3 2 4d2277f46a699d475ff095fa274a007fdf8281ad8bccb3575feb62779e257e8e wrath
IN SSHFP 4 1 59c3ed21e086b923a4e8a49504691c844f5a1590
wrath IN SSHFP 4 2 4e2f1c2ee4850d1bb43fffd43e16d27df99d0a3491582f51423dd7d48944f513
```

Load these records into your DNS server.

You could also copy the server's public key files to a central host and tell `ssh-keygen` to use those files with the `-f` flag.

```
$ ssh-keygen -r wrath -f ssh_host_ed25519_key.pub
```

You must run this command separately for each key file, but if you have a central automation server this approach has a lot to recommend it. Remember, the public keys are displayed to anyone who can connect to the server's SSH port. Copying the public key files to a secure server is not usually a security risk.

As I write this some free DNS providers, such as Hurricane Electric, support SSHFP records.

Configuring the Client

The OpenSSH client might use SSHFP records by default, depending on how the operating system distributor compiled it. Use the VerifyHostKeyDNS keyword to explicitly define what `ssh` should do. If set to `yes`, the client completely trusts keys provided by SSHFP records. If set to `ask`, `ssh` displays the key fingerprint

and asks the user what to do.

This handles the host keys. Now let's talk user authentication keys.

Distributing authorized_keys

A lone sysadmin with only a handful of servers can pretty easily maintain her own *authorized_keys* file. Get up to seven or eight servers, and copying *authorized_keys* everywhere gets pretty tedious. Have a whole team of sysadmins, and want to ban password authentication across your hundreds of servers? You really have to look at ways to automate *authorized_keys* replication. You can either have your automation system replicate authentication keys on all systems, or have sshd query the network for a user's *authorized_keys* at every login attempt. Both have their place.

Replicating Key Files

Having users maintain their own key files can cause operational problems. Users have an uncanny ability to corrupt their files, especially when they think they know what they're doing. By having a centralized system to deploy *authorized_keys*, you get a chance to perform some basic integrity tests before the user gets themselves in trouble. You don't need a complicated key file parsing and validation system, but being able to say, "Did you realize that your key entries have newlines in the middle of them?" can reduce annoyance for everybody involved. Also, if a user's workstation gets hacked into, and the intruder bootstraps that into server access, the intruder can add their own key to the user's *authorized_keys* and copy it to all the servers in *known_hosts*. Centralizing key management and removing a user's ability to upload new key files without passing through the automation system can be desirable.

You really don't want your automation system mucking around in each user's home directory. Instead, take advantage of the AuthorizedKeysFile *sshd_config* keyword. This lets you put a user's *authorized_keys* file anywhere you want. Combine this with the %u token to have **root** own all the user keys.

```
AuthorizedKeysFile /etc/ssh/keys/%u Remember that the %u token represents the username.
With this AuthorizedKeysFile setting, the authentication keys for the user mwl would be in
/etc/ssh/mwl, while the keys for the user djm would be in /etc/ssh/djm. Key files outside
the user's home directory look exactly like any other authorized_keys, but they must be
owned by root. Even if our hypothetical intruder penetrates an account, they can't edit
the keys without privilege escalation.
```

Use any features your operating system supports to secure these files. On a UFS filesystem, maybe the immutable flag would suit your environment. Or NFSv4 ACLs. If something annoyed you by refusing to let you change a file, consider it for protecting authorized key files.

Querying the Network for Keys

If you have centralized authentication system such as LDAP, you can store user

authentication keys in that system. OpenSSH can query that information source with the AuthorizedKeysCommand and AuthorizedKeysCommandUser keywords.

```
AuthorizedKeysCommand /usr/scripts/getAuthorizedKeys.pl AuthorizedKeysCommandUser ldap Any time you look at network-based authentication people’s brains leap into LDAP. LDAP is specifically meant for this sort of directory lookup—it’s pretty much a database optimized for reads. I can’t go into detail here, as LDAP directories vary wildly between vendors. No matter which you use, however, you’ll need to get an SSH key schema loaded into your directory. Talk with your LDAP administrator and see what they can provide. The exact schema needed varies with the directory arrangement, but it usually involves attaching an sshPublicKeys entry to the user’s account. LDAP administrators for large enterprises that are built upon commercial LDAP offerings are often reluctant to extend core directory entries, because that limits their ability to get vendor support. In my experience, solving this problem required more effort than any other part of key distribution.
```

Once you have the schema loaded, you need a script to fetch *`authorized_keys`* from the directory. The type of script varies precisely as much as the types of authentication systems people use. A script that authenticates against Active Directory will be completely different from one that authenticates against a home-brewed OpenLDAP directory. CentOS ships with a script to authenticate against their LDAP server, ssh-ldap-helper(8). People have solved this problem for a variety of directory services, and made their scripts available, so be sure to look for existing solutions before spending the next ten years debugging your own.

The AuthorizedKeysCommandUser keyword defines the account that will run the script in AuthorizedKeysCommand. If you don’t set AuthorizedKeysCommandUser, `sshd` will not run the script. All attempts to get a user’s *`authorized_keys`* will fail. I recommend creating a user with no privileges except running this one script. Isolated unprivileged users are a ridiculously inexpensive security solution that doesn’t get used often enough.

Just because LDAP gets all of the attention, don’t limit yourself by thinking LDAP is a requirement. It’s convenient if you have it, yes, but you can use any service that makes sense for your environment. If your organization has a rule that all applications must interoperate via ODBC, or perhaps Wordpress XMLRPC over HTTPS, leverage your existing expertise and write a script that fetches keys that way. AuthorizedKeysCommand is a script. You’re a sysadmin. This is your thing.

Whether you’re talking about user authentication keys or host public keys, automation and key distribution are vital. Now that you can have your automation manage SSH, let’s see how SSH can manage automation.

---

[1] And remember, your users are quick—especially when it’s inconvenient.

[2] Emailing a Registry file to all of your users and telling them to double-click on it before using SSH does not encourage a security mindset.

Chapter 12: Automation

SSH is an incredibly powerful tool for automation. Many programs can use SSH as a transport, relying on known-secure software rather than attempting to implement their own network security. Most network orchestration tools like Ansible and Puppet use SSH; breaking your SSH configuration means you can't use them.

This same flexibility can cause security issues, however. Automated processes should not get access to anything except the bare minimums needed to perform their task. Fortunately, you can limit the commands that particular users can run via SSH, through the `authorized_keys` file or even in `sshd` itself. Additionally, you can automatically run commands whenever a user logs in. We'll start with that function, and proceed to limiting users.

Running Commands at Login

The SSH server checks for commands to run any time a user starts a new session. This was mostly designed to configure services needed to make the account usable before login, such as mounting filesystems and assigning an X display, but you can use it for whatever you need.

At login, `sshd` checks for the shell script `$HOME/.ssh/rc`. If it exists, it gets run. If it doesn't exist, `sshd` checks for a script at `/ssh/sshrc` and runs that. Either way, the script is run by the account being logged into. If you need to perform tasks every time a user logs in, consider this functionality.

The script must be a valid shell script, complete with `#!/bin/sh` at the top of the file, and it must be executable. (Some Linux distributions execute this command even if it doesn't meet these requirements.) The SSH daemon hands the script one argument, an X11 cookie. With modern X software, you almost certainly can ignore it.

The `sshd_config` keyword PermitUserRC turns this script check on and off. While it defaults to `yes`, you can disable the script by setting it to `no`.

authorized_keys Restrictions

While a user's `authorized_keys` dictates the key pairs that can be used for authentication, you can also use it to limit the commands that a user logged in with that key may run. One account might have a key pair for interactive use and a second key pair for an automated task. Configuring requires understanding the `authorized_keys` file format.

Authorized_keys Format

A minimal `authorized_keys` entry has three parts: the key type, a few hundred

alphanumeric characters representing the public key, and a comment field. Each entry goes on a single line, no matter how long it is. It will look something like this: ssh-rsa AAAAB3NzaC1yc2EAAAABJQAAA… wE2Ime8Rs/Q== moose-20160525

This is an RSA key, as shown by ssh-rsa at the beginning of the entry. This public key begins with a AAA and ends with 8Rs/Q==. Many but not all public-key entries end in the double equals sign. The comment at the end gives the host this key was created on and the date of creation.

You can put additional keywords and instructions on how this key may be used at the beginning of the entry. The server obeys those instructions, within the limits of the user's permissions. Find a complete list of *`authorized_keys`* keywords in the sshd(5) man page, but here are the most commonly used ones.

command="command"

Whenever someone logs in using this key, run the specified command. SSH ignores any command provided by the user in favor of the one dictated by *`authorized_keys`*. You might use this for automated processes, such as configuring a VPN (Chapter 13) or running `rsync`.

```
command="sudo ifconfig tun0 inet 192.0.2.2/30" ssh-rsa AAAAB3NzaC1yc2EAAAABJQAAA…
```

One interesting feature is that SSH retains any command the client requested in the environment variable `$SSH_ORIGINAL_COMMAND`. You can have *`authorized_keys`* run a script that checks this environment variable and acts appropriately. ("The backup account just requested access to */bin/bash*? Hello, sysadmin, we have a problem…")

environment="NAME=value"

This set an environment variable when this key is used to log in. You can use any number of environment statements.

```
environment="automated=1" ssh-rsa AAAAB3NzaC1yc2EAAAABJQAAA…
```

By default, `sshd` does not permit setting environment variables. The sysadmin must set PermitUserEnvironment to `yes` in *`sshd_config`* for users to set environment variables.

from="ssh-pattern"

This key can only be used for authentication if the clients address or reverse DNS matches the given pattern. We discussed patterns in Chapter 2. I frequently use this to restrict automated processes. Even if an intruder steals a private user key, he cannot access the SSH server from any host other than the one I permit.

```
from="198.51.100.0/29" ssh-rsa AAAAB3NzaC1yc2EAAAABJQAAA…
```

Only hosts in the IP range 198.51.100.0 through 198.51.100.7 can use this key to log into the SSH server.

You can only use host names in the pattern if UseDNS is set to `yes`.

Remember that intruders can frequently forge their reverse DNS entries, so it's most often best to disable DNS in `sshd` and stick with IP addresses.

no-agent-forwarding

This disables SSH agent forwarding (see Chapter 7) for this key.

```
no-agent-forwarding ssh-rsa AAAAB3NzaC1yc2EAAAABJQAAA…
```

no-x11-forwarding

This (wait for it…) disables X forwarding (see Chapter 8).

```
no-x11-forwarding ssh-rsa AAAAB3NzaC1yc2EAAAABJQAAA…
```

no-pty

Sessions that authenticate with this key will not be granted a pseudo-terminal. Many programs that run under automation do not need a terminal.

```
no-pty ssh-rsa AAAAB3NzaC1yc2EAAAABJQAAA…
```

no-user-rc

This disables `sshd`'s login script checks, as discussed in "Running Commands at Login" at the beginning of this chapter.

```
no-user-rc ssh-rsa AAAAB3NzaC1yc2EAAAABJQAAA…
```

permitopen="host:port"

The permitopen keyword restricts local port forwarding so that it can only attach to the given hostname or IP address and port on the local machine. If the server doesn't allow local port forwarding, this has no effect.

```
permitopen="localhost:25" ssh-rsa AAAAB3NzaC1yc2EAAAABJQAAA…
```

This example allows port forwarding to connect to port 25 on 127.0.0.1, but nothing else.

You can set permitopen to `none` to disallow all port forwarding.

tunnel="n"

Use a specific tunnel device number for SSH tunnels (see Chapter 13).

```
tunnel="3" ssh-rsa AAAAB3NzaC1yc2EAAAABJQAAA…
```

restrict

By default, anything not denied is permitted. The restrict keyword inverts that, blocking everything unless you specifically allow it. You can use the keywords agent-forwarding, port-forwarding, pty, user-rc, and X11-forwarding to turn those functions back on.

Using Multiple Keywords

As with just about everything in OpenSSH, you can use multiple keywords in one entry. Separate keywords with commas, not spaces.

```
restrict,command="/usr/local/scripts/backup.sh" ssh-rsa AAAAB3NzaC1yc2EAAAABJQAAA…
```

Keys and Automated Programs

Lots of us want to use SSH as a secure transport for other programs. Maybe you have a custom monitor program, or a backup process that runs over rsync. Such clients should never have a hard-coded username and password; in addition to

being insecure, it's neither maintainable nor scalable. One solution is to use an authentication key without a passphrase. By tightly restricting how that key can be used and what actions can be taken with that key, you minimize the damage an intruder can inflict.

Note that potential damage is only minimized, not eliminated. An rsync backup run at the wrong time can damage an existing good backup or saturate the network. Bringing a VPN up at the wrong time can be highly disruptive. In most environments, however, these are less damaging and more visible than someone copying or destroying all of your proprietary data.

First you need a user key suitable for use by a program, then you need appropriate *authorized_keys* restrictions.

Automation Authentication Keys

Automated processes cannot type passphrases. Any scheduled or otherwise automated task that requires SSH access to another host needs a key without a passphrase. Generate this key exactly like you would generate a host key.

```
$ ssh-keygen -f filename -N ''
```

This creates two files, one with your chosen file name and one with that same name but *.pub* appended. Here I create a key called *task-key*.

```
$ ssh-keygen -f task-key -N ''
```

I end up with the files *task-key* and *task-key.pub*. The *.pub* file is the public key.

Either create an account on the SSH server for this automated task, or choose an existing account. The host's SSH server must permit logins to that account. Add the *.pub* file to that account's *authorized_keys*.

The client machine should now be able to log on to the SSH server using the key. Remember to use the `-i` argument to ssh(1) to specify the alternate key file. Here I use this key to log on to the machine **sloth**.

```
$ ssh -i task-key sloth
```

If you successfully log onto the server, the key is correctly installed. Now let's lock it down.

Limiting Automation Keys

Best practice forbids all access unnecessary for a user to perform his task. Does your automated process need port or X forwarding? Turn them off. Does it need a special environment? Probably not, because you can establish that environment more easily in the user account. Your automated job runs on a single machine, so you can restrict the key so that it can only be used from that one machine. You'll probably end up with an *authorized_keys* entry like this.

```
restrict,command="dump /home > /backups/`date +s`.dump",from="192.0.2.8" ssh-rsa
AAAAB3NzaC1yc2EAAAABJQAAA…
```

Configuring the key like this reduces the scope of disasters. The backup script won't accidentally overwrite your root partition. An intruder can only run

your backup script. This isn't great, but it's better than the intruder stealing your data and deleting your log files.

Developing Automation Scripts

One challenge in restricting a key is understanding what commands the program actually needs, versus what you think it needs. The debugging mode of `sshd` can help you figure this out. Have your client run its command against `sshd` in debugging mode, and study the output. You'll see all of the commands that the client runs. This will also let you lock down the key further than you might otherwise—if you know the exact flags rsync will use on your server, you can impose those as a restriction.

I've written scripts that seem to work from the command line, but fail when scheduled, and each time it's turned out that my script was picking up authentication information from my SSH agent rather than using the key I'd created for the task. The IdentitiesOnly keyword tells ssh(1) to only use the identity specified on the command line and not your agent. Set `-o IdentitiesOnly=yes` in your script's SSH command.

An automated script should never be prompted for user input. You don't want your script hanging and waiting at a password prompt. The BatchMode keyword disables password and passphrase prompts. By setting BatchMode to `yes` the SSH part of your script will crash and die immediately, rather than pointlessly hanging around forever.

Server-Side Restrictions

Perhaps you don't want to use *`authorized_keys`* to restrict access, or maybe you'd like additional protections. You can use the ForceCommand *`sshd_config`* keyword to restrict what an account can run.

ForceCommand takes one argument, the command to be run. It's run under the user's regular privileges, and disregards whatever command the client requested. Much like defining a command in *`authorized_keys`*, ForceCommand retains the requested command in the `$SSH_ORIGINAL_COMMAND` environment variable.

ForceCommand is best used inside a Match statement.

Automation and Root Logins

"My command needs to run as root!" It is possible to login as root using the PermitRootLogin keyword. Don't do it. Logging in as root for automation breaks many fundamental security principles. Trusting your automated scripts with remote root privileges is a good way to spend an unscheduled weekend restoring the servers from backup. (You do have backups beyond rsync, right? Remember that rsync is a tactic, not a strategy.) Yes, a few environments can securely support root logins. Some people are using root logins in a manner that can

support auditing. If you're reading this book to learn about SSH, however, your environment is nowhere near ready for this.

If your automated process needs privileged access, use sudo. Sudo (https://www.sudo.ws) lets unprivileged users run particular commands with elevated privileges and is available for every Unix-like system. I'm not going to go into detail on using sudo; if you need a tutorial, check any number of websites or my book *Sudo Mastery* (Tilted Windmill Press, 2013). Sudo is far more flexible, and more dangerous, than most people give it credit for.

We'll use restricted keys in the next chapter to build a VPN over SSH.

Chapter 13: Virtual Private Networks

You can wrap SSH around arbitrary TCP connections, adding a layer of encryption to any protocol. But OpenSSH also supports building generic tunnels that can pass all traffic and all protocols, not just TCP. You can link to remote offices with OpenSSH, creating a Virtual Private Network (VPN) that allows users at one office to access the other office almost as if they were on the next floor rather than the next country.

VPNs are an OpenSSH extension to the SSH protocol. PuTTY does not include VPN functions and the PuTTY developers have repeatedly stated that they do not intend to add it to their client (see the *tun-openssh* wish list item on the PuTTY website). We will only examine OpenSSH VPNs on Unix-like systems.

SSH was not designed as a generic VPN protocol, and tunneling protocols inside TCP is terrible practice. When a TCP connection loses packets, it must re-transmit those packets until the other end of the connection acknowledges receipt. By wrapping a TCP connection inside another TCP connection, you amplify the effects of packet loss. TCP-based VPNs collapse in the face of congestion. I strongly recommend using OpenVPN instead of OpenSSH for your VPN. An OpenSSH VPN does have the advantage that it only requires a single TCP port open between the client and the server. If that's all the connectivity you have, an OpenSSH VPN might be your least terrible option.

A VPN is perhaps the most complicated thing you can do with OpenSSH. This chapter assumes you are comfortable with the earlier chapters, including public-key authentication, keeping an SSH session alive, and restricting the commands available to SSH clients.

Example Network

Our SSH client, `avarice.mwl.io`, has two network interfaces. One is on the public Internet. While we could refer to that interface by IP address, we'll use the hostname instead. The second interface is on private network A, with an address of 172.16.0.1/24.

The SSH server, `gluttony.mwl.io`, also has one interface on the public Internet. We'll refer to this interface by hostname rather than IP. Its second network card is on private network B, and has an IP address of 172.17.0.1/24.

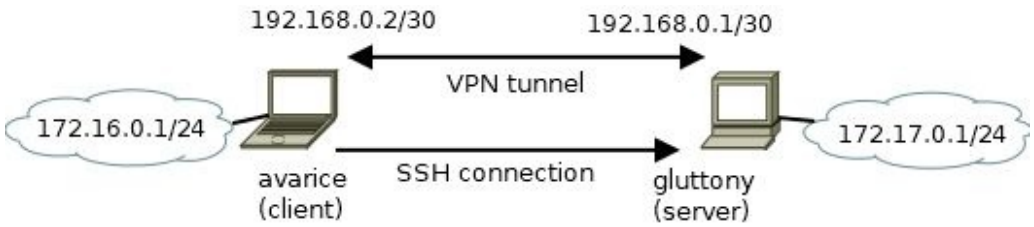


Figure 13-1: VPN Network We'll use SSH to establish a point-to-point tunnel between the two hosts. The client's end of the tunnel will have the IP address 192.168.0.2/30. The server end of the tunnel gets 192.168.0.1/30.

We'll consider OpenBSD, FreeBSD, Debian, and CentOS. OpenBSD has the best SSH VPN support of any operating system—which shouldn't surprise anyone, considering that OpenSSH originates in OpenBSD. Running an SSH VPN on FreeBSD requires basic scripting. Most Linux distributions change OpenSSH to fit better with their systems, and they've also deprecated the standard UNIX networking commands in favor of Linux-specific tools. This means every operating system needs a different approach. Between these four, you should find a method that you can adapt for your operating system.

Creating and managing VPNs is the most difficult feature in OpenSSH, and the operating systems that support them change over time. I wouldn't be shocked to see these VPN instructions become outdated more quickly than the rest of this book. If you have trouble with these examples, consult your operating system documentation for more current references.

Common Concepts

The following concepts and configurations for OpenSSH VPNs appear across all operating systems. No matter which OS you run, you must understand this material and follow these general principles. While you can find tools that purport to simplify tunnel setup, once you understand how the tunnel works you'll find using raw SSH trivially simple.

Tunnel Interfaces

An SSH VPN works using a tunnel (or *tun*) interface. A tunnel is a virtual interface that sits above some other network interface. The most common use for tunnel interfaces is to create a virtual link between two separate hosts, such as in a VPN. This tunnel is treated as a point-to-point connection. The method for creating tunnel interfaces varies by operating system.

When you use an SSH VPN, the client and server both attach themselves to tunnel interfaces on their respective machines. When the operating system sends a packet to the tunnel, the packet is relayed through the SSH connection. When the other machines SSH process receives the packet, it unwraps it and sends it to the operating system via the local tunnel interface.

Just like any other interface you want to use for IP routing, your tunnel

interfaces need IP addresses. You must route traffic destined for the remote network to the IP address at the remote end of the tunnel. We'll demonstrate this in each example.

Each tunnel interface needs a device number, like any other device on a Unix-like system. Just as your network interface might be eth0 or em1, tunnel devices might tun0 or tun1. Our examples use device zero, creating device names like tun0. If you have many tunnel devices I recommend both assigning a specific device for each purpose and reassessing your design choices.

SSH Server Configuration

The `sshd_config` keyword PermitTunnel specifies if a client may establish a VPN tunnel. PermitTunnel has four valid options: yes, no, point-to-point, or ethernet. If set to `no` (the default), tunnels are forbidden. If set to `yes`, all tunnels are permitted.

A point-to-point tunnel is a virtual private circuit that runs from one spot to another. A point-to-point tunnel requires routing to be usable. This is usually the best type of tunnel for an SSH VPN.

```
PermitTunnel point-to-point An Ethernet tunnel transmits layer 2 traffic, permitting two
separate locations to share their local LAN. Don't tunnel Ethernet over SSH if you can
possibly avoid it. Local network problems on one side of the VPN can propagate across the
link and saturate your external bandwidth. SSH VPNs are already vulnerable to congestion;
don't amplify that problem even more.
```

To use an SSH VPN, the SSH processes must have sufficient privileges to make changes to the tunnel devices and the routing table on both the client and the server. Creating an SSH VPN requires **`root`** privileges on both the client and the server. You'll run `ssh` as **`root`** and log in directly as **`root`**. I stated earlier that logging in as **`root`** is a terrible option. I stand by that statement. If you're using an SSH VPN, however, you're basically out of good options.

Here I permit our SSH client a root login on the SSH server, but only through public-key authentication. I also allow that IP address to open a tunnel.

```
Match Address avarice.mwl.io PermitRootLogin prohibit-password PermitTunnel point-to-point
In your production configuration, use the client's IP address rather than the hostname.
```

Very old versions of OpenSSH might not let you put the PermitTunnel statement inside a Match statement. If you encounter such an `sshd`, immediately upgrade the server's OpenSSH—it's not safe to have on the public Internet.

IP Forwarding

For an SSH VPN to connect two different networks, both the SSH server and the client must forward packets from one interface to another. This is called *IP forwarding*. Forwarding packets between interfaces is the only difference between a host and a router. The SSH client receives packets on its internal Ethernet interface, and transmits those packets meant for the remote location across the VPN. Similarly, the SSH server accepts packets bound for the other

office on its internal interface and shoots them across the VPN.

VPN Authentication Key

Use key authentication with VPNs. If you're going to bring up your VPN manually, only on special occasions, create a standard user authentication key as discussed in Chapter 7, "SSH Keys". If an automated process will start the VPN, create a key without a passphrase as covered in Chapter 12, "Automation". Put the key in a special file, such as `/root/.ssh/tunnelkey` on the client.

Copy the key's public key to the server's `/root/.ssh/authorized_keys`. This key should only be able to run the VPN commands; even with key-based authentication, you don't want a remote intruder able to get a root login on your server. Chapter 12 discusses restricting key privileges, but the exact commands needed vary by operating system.

The SSH Tunnel Command

Activate an OpenSSH tunnel with the `-w` flag.

```
# ssh -i keyfile -f -wclientTunnelNumber:serverTunnelNumber servername true
```

The `-i` tells `ssh` which private key file to use. The `-w` tells the client to request a tunnel, and which tunnel device numbers to request on each side. The `-f` puts `ssh` into the background, so that you don't have a command prompt on the remote system. And we run true(1) just so we have a command that always runs successfully.

In our examples, the key file is `/root/.ssh/tunnelkey`. I want to use tunnel device 0 on each side, and the server is **`gluttony.mwl.io`.**

```
# ssh -i tunnelkey -f -w0:0 gluttony.mwl.io true
```

If all works well, this should silently return to a local command prompt.

Some of these command-line options can be set in `ssh_config`. I recommend placing tunnel options in `/root/.ssh/config`, rather than the system-wide configuration. You don't want an unprivileged user's innocent SSH session attempt to open a tunnel and route across it.

```
Host gluttony.mwl.io
Tunnel point-to-point
TunnelDevice 0:0
IdentityFile /root/.ssh/tunnelkey IdentitiesOnly yes
```

Add other options for the host as your environment or the operating system requires. This strips down the command line needed to activate the tunnel.

```
# ssh -f gluttony.mwl.io true
```

Our examples assume that you have enabled root logins, copied the client's public key to the server, and set up the host's key and tunnel devices in `/root/.ssh/config`.

Debugging

If you follow the steps for your operating system and the tunnel doesn't start, run `ssh` in verbose mode. You'll see the details of your errors. If that doesn't help, run

`sshd` in debug mode. Search the Internet for the exact text of your error messages. You will certainly find people who have experienced and solved your problem.

Now let's configure some VPNs.

OpenBSD

OpenSSH is developed inside OpenBSD, and the OpenBSD team created the OpenSSH VPN function, so OpenBSD has very good support for OpenSSH VPNs. Start by tightening up what your client may access with this key by putting controls in *`/root/.ssh/authorized_keys`*.

```
restrict,tunnel="0",command="/bin/sh /etc/netstart tun0" ssh-rsa AAAAB3Nza…
```

I've locked down all the key-based options, then added the ability to access a specific tunnel device and run the command that configures that tunnel. Even if the client is compromised and logs into the server as **`root`**, it can't inflict much damage.

Enable packet forwarding on OpenBSD by setting the sysctl net.inet.ip.forwarding to 1.

```
# sysctl net.inet.ip.forwarding=1
```

To make this change permanent across reboots, make a matching entry in *`/etc/sysctl.conf`*.

```
net.inet.ip.forwarding=1
```

Now configure your tunnel devices. You'll need an *`/etc/hostname.tun0`* on both the client and the server. Each contains two lines. Here's the client: 192.168.0.2 192.168.0.1 netmask 255.255.255.252

```
!route add 172.17.0.1/24 192.168.0.1 > /dev/null 2>&1
```

The first line creates a tunnel interface with a local IP of 192.168.0.2 and a remote IP of 192.168.0.1. OpenBSD will configure this interface at boot, but the interface won't be active; the tunnel isn't attached to anything. When you activate your SSH tunnel, it attaches to the tunnel interface. The second line of *`hostname.tun0`* is a command that OpenBSD runs when the tunnel activates. This command configures routing to the LAN behind the server.

The server's *`hostname.tun0`* looks really similar.

```
192.168.0.1 192.168.0.2 netmask 255.255.255.252
!route add 172.16.0.1/24 192.168.0.2 > /dev/null 2>&1
```

The IP addresses are reversed. When the tunnel comes up, the network behind the client gets routed across it.

SSH from the client to the server. The tunnel should come up and configure itself.

FreeBSD

FreeBSD doesn't incorporate OpenSSH VPNs out-of-the-box, but they're really easy to set up. The easiest method is via calling a shell script when the tunnel comes up. You can avoid that need by being tricky and clever, but tricky and

clever has an uncanny ability to bite you during an outage. Additionally, I'll use the scripts to illustrate a couple OpenSSH features.

First, enable packet forwarding on both the client and the server. Use the sysctl net.inet.ip.forwarding as in OpenBSD, or set GATEWAY_ENABLE=YES in */etc/rc.conf*.

```
# sysrc gateway_enable=YES
```

Now let's get the scripts ready. The server will use the script */usr/local/scripts/tunnelserver.sh*. I'll lock the client's entry in *authorized_keys* to permit it to run only that script.

```
restrict,tunnel="0",command="/usr/local/scripts/tunnelserver.sh" ssh-rsa AAAAB3Nz…
```

Whenever this key is used to log in, `sshd` runs the configured script. Let's look at the server-side script.

```
#!/bin/sh
/sbin/ifconfig tun0 192.168.0.1/30 192.168.0.2
/sbin/route add -net 172.16.0.0/24 192.168.0.2
```

The script adds IP addresses to the tunnel interface and configures a route to the remote network.

We'll use a similar script on the client, */usr/local/scripts/tunnelclient.sh*, to add the IP addresses and routes to this side of the tunnel.

```
#!/bin/sh
/sbin/ifconfig tun0 192.168.0.2/30 192.168.0.1
/sbin/route add -net 172.17.0.0/24 192.168.0.1
```

SSH-ing into the server activates the tunnel and configures the server side of it. You'll need to run the client script to configure the client side. Fortunately, ssh(1) has the LocalCommand keyword to automatically run a command when you connect to a host.

```
Host gluttony
Tunnel point-to-point
TunnelDevice 0:0
IdentityFile /root/.ssh/tunnelkey IdentitiesOnly yes
PermitLocalCommand yes LocalCommand /usr/local/scripts/tunnelclient.sh Here I use
PermitLocalCommand to say “yes, you may run a command locally when you connect,” and
LocalCommand to define the command.
```

When you SSH from the client to the server, the tunnel should come up automatically.

CentOS and Debian

Configure SSH VPNs on these two popular Linux distributions in a very similar way to FreeBSD. Both distributions have obsoleted standard Unix tools like ifconfig(8) and route(8), however, so we must use the Linux-specific ip(8) instead.

Create your tunnel key, enable SSH tunneling, and permit root logins using keys, as discussed in "Common Concepts." Then the client needs a */root/.ssh/config* precisely like that used for FreeBSD.

```
Host gluttony
Tunnel point-to-point
```

```
TunnelDevice 0:0
IdentityFile /root/.ssh/tunnelkey IdentitiesOnly yes
PermitLocalCommand yes LocalCommand /usr/local/scripts/tunnelclient.sh Enable IP
forwarding on Linux setting the sysctl net.ipv4.ip_forward to 1 in /etc/sysctl.conf.
Debian already has this entry, commented-out.
```

Lock down the server's *`/root/.ssh/authorized_keys`* so that this key can only open the tunnel device.

```
restrict,tunnel="0",command="/usr/local/scripts/tunnelserver.sh" ssh-rsa AAAAB3Nz…
```

Now all you need are the Linux scripts. Here's a *`/usr/local/scripts/tunnelserver.sh`*: #!/bin/sh

```
ip addr add 192.168.0.1/30 dev tun0
ip link set dev tun0 up ip route add 172.16.0.0/24 via 192.168.0.2 dev tun0
```

And here's a client script.

```
#!/bin/sh
ip addr add 192.168.0.2/30 dev tun0
ip link set dev tun0 up ip route add 172.17.0/24 via 192.168.0.2 dev tun0
```

Now run `ssh -f gluttony` from your client, and your tunnel will come up.

With these three examples, you should be able to get an SSH VPN running on any Unix-like operating system. Remember that an SSH VPN is not a wonderful solution, though. Before you have trouble or experience congestion, investigate real VPN software like OpenVPN.

Chapter 14: Certificate Authorities

The hardest part of using SSH correctly isn't the software, or obscure or hidden checkboxes, or even more obscure command-line arguments. It's verifying keys. Users are expected to verify host keys, a tedious process that most of them won't even bother with. Users generate authentication keys, but then they need to be copied around the network. If you're managing your systems with automation, you can automate part of the verification process and vastly reduce risk by implementing an OpenSSH Certificate Authority.

An SSH CA is not the same as the X.509 Certificate Authority you're probably familiar with from the TLS[1] deployed on websites. If you had to purchase an X.509 certificate for each and every host to use SSH, you wouldn't bother. But take a moment and consider what a certificate authority does for you.

A certificate authority is a method of delegating trust. Every web browser has a list of trusted certificate authorities built into it. When your browser calls up a website that uses a certificate, the browser checks to see if that certificate is signed by a trusted certificate authority. If it is, the browser trusts the certificate on that website. If the certificate is signed by anything other than a trusted certificate authority, the user sees a warning.

SSH public keys resemble self-signed certificates. The server is declaring, "this is who I am, and you can either accept this or go away." You create an SSH CA by giving your clients and servers a certificate that they trust. Install this certificate on all of your OpenSSH software, and it will trust any public key signed by that CA key.

X.509 certificates are complicated in part because they're part of a global network of certification. Organizations use TLS certificates to secure websites, email, and pretty much any other arbitrary TCP connection. Certificates contain fields for a whole bunch of stuff that most of us will never need.

SSH certificates only need the ability to digitally sign data and carry a few chunks of metadata. They're not a global entity. An OpenSSH CA is entirely internal. A standard SSH key pair has all the functionality needed to sign keys.

Once you deploy an OpenSSH CA, clients configured to trust the CA key will automatically trust host keys signed with that key. Servers configured to trust that CA key will automatically trust user authentication keys signed with the CA key. Your users will only see warnings when they connect to hosts outside of your organization, or if something is seriously wrong.

Certificate authorities are an OpenSSH extension. Other clients have not yet adopted them. Even if all of your desktop clients run something other than

OpenSSH, SSH certificate authorities are useful for verifying host keys when connecting between servers.

Don’t even consider deploying an OpenSSH CA unless you can automatically distribute files to all your servers and remotely restart `sshd` on them. If you don’t have automation in place, take a look at Ansible or one of its competitors.

An SSH CA has a whole bunch of functions that are only useful in edge cases. An organization like Google or Facebook needs a whole bunch of features that most of us don’t. Read the ssh-keygen(1) man page for the full range of CA options. Here we’ll set up a comparatively simple CA for a middle-sized network, starting with the simpler host certificates, then proceeding to the more complex user certificates.

Certificate Expiration

A critical part of signed certificates is that they expire. Yes, you could set all of your certificates to be good for a quarter-century, but those keys will be insecure long before that. Plan from the beginning to use your automation system to regularly update your certificates.

Attacking public keys computationally, in the absence of a flawed implementation, can take billions of years. Software has bugs, though, and it’s possible that a bug might let an intruder crack a key in much less time. Also, those aeons needed to computationally break a key are averages. The intruder might get lucky. Eternally valid certificates increase the intruder’s chances of success.

How long should a certificate be good for? Rolling over certs every year or so is most common. If you have one of those orchestrated networks where servers appear and disappear by the magic of automatic deployment, you might want to regenerate your host key certificates every week or month.

In short, never plan to use certificates longer than a year, except in those rare cases where a host cannot be changed. No, I don’t mean “the boss would really like this host to never change,” I mean “the federal government has declared this host a life-sustaining service and changing the certificate is a felony that carries a minimum jail sentence.”

Don’t set your certificates to expire in exactly one year, though. Remember, life happens. Maybe you put on your calendar to renew all of your certificates in 52 weeks, but you develop appendicitis the day before and you’re off work for three weeks. I allow at least a month of leeway for such emergencies, so these examples assume we expire all certificates in fifty-six weeks and five days.

Better still? Use your automation system to renew and replace all certificates at half their expiration date. Issue and deploy new one-year certificates every six

months. As you gain confidence in your automation and work out the bugs, slowly decrease that time. Automatically create new certificates every year, then every month, then every week. Then reduce the time you use your CA keys. Making certificate renewals painlessly routine can transform potential key compromises from disasters into trivialities.

SSH CA Keys

Before you even think about creating an OpenSSH CA, consider how you're going to handle and secure those keys. Your certificate authority is the key to your kingdom. Protect it as you would any other critical infosec asset. An intruder that compromises your OpenSSH CA can create user keys trusted by all of your servers. That would be bad. I keep my OpenSSH CA on a dedicated-purpose OpenBSD machine that only gets booted when I need to sign keys. Larger organizations will want to put their OpenSSH CA in the same part of the network where they keep other, similarly critical hosts.

Best practice recommends creating two certificate authorities: one for certifying host keys, the other for user keys. Different teams of people manage users and hosts, and having two different certificate authorities allows each to use the workflows best suited to those tasks. Each CA might even reside on different machines, in different parts of the network. While you can install any number of certificate authorities on a host, making it possible to split CAs later, very few sysadmins regret complying with such a simple best practice from the beginning.

As your network grows, so will the number of public keys you manage. Organize your CA well from the beginning to minimize later struggle.

I recommend putting your CA in a directory like `/usr/local/sshca`. Create subdirectories for host and user keys, with unambiguous names like `/usr/local/sshca/hosts` and `/usr/local/sshca/users`. Each host and user should get its own subdirectory therein, such as `/usr/local/sshca/hosts/sloth` and `/usr/local/sshca/users/mwl`. Don't put your CA in `/root`, and especially not `/root/.ssh`. Those directories are for the **`root`** account's information, much as `/etc/ssh` is reserved for this particular host's SSH services. [2] A certificate authority is a major project, and deserves its own directory.

Why separate by directory and not by filename? Each host and user has files with the same name. You'll find `/etc/ssh/ssh_host_ecdsa_key` on every single SSH server, while every user has an `id_rsa.pub`. While it's certainly possible to copy that file on the host **`sloth`** to `sloth-ssh_host_ecdsa_key`, generate a cert for it, and then rename the cert as you're sending it back to the server, that's a couple extra steps. Giving each host and user unique directories decreases fragility.

Create an SSH CA key the same way you would manually create a host key. I add the `-c` flag to add a special comment to the key. SSH CA keys look like every other SSH key, so the comment helps identify them. Here's a host key signing key.

```
# ssh-keygen -t rsa -f host-mwlca-key -c 'CA host key generated 2017-11-30'
```

Use a good passphrase. You'll be able to use your SSH agent for mass signings, so feel free to make it complex. You'll get the file *host-mwlca-key* containing the private key for the certificate authority, and the file *host-mwlca-key.pub* with the public key.

Creating a user-certifying CA is exactly the same, except for the file name and the comment.

```
# ssh-keygen -t rsa -f user-mwlca-key -c 'CA user key generated 2017-11-30'
```

Protect these private keys. Feel free to spam your whole network with the public keys, however.

Trusting Your Certificate Authority

The ssh(1) client and sshd(8) server have completely different ways of configuring certificate authorities.

sshd(8) and Certificates

SSH servers use user certificates to validate certificates used for authentication. Set a file containing all of your trusted certificates in *sshd_config*, using the TrustedUserCAKeys keyword.

```
TrustedUserCAKeys /etc/ssh/user-ca-keys.pub The CA file contains one CA public key per line, and accepts comments marked off with a leading pound sign (#). It looks exactly like an authorized_keys file.
```

Restart sshd, and it will trust keys signed by this certificate authority.

ssh(1) and Certificates

SSH clients use host certificates to validate host public keys. Configure trusted host certificate authorities in *known_hosts*. The most effective place for a CA key is in */etc/ssh/ssh_known_hosts*, both so that all clients immediately recognize the CA and so users can't muck with the key.

Don't just copy the CA's public key file to *known_hosts*, though. You must mark this key as a certificate authority and add the hostnames this key is valid for. Copy the public key to a separate file—never muck with your original key files! Add the marker `@cert-authority` to the beginning of the line, then add an SSH pattern for the hosts this key is valid for. This key is valid for all hosts in the **mwl.io** domain.

```
@cert-authority *.mwl.io ssh-rsa AAAAB3NzaC1yc2EAAAADAQABA…
```

If the key is valid for multiple domains, separate them with commas. Don't use spaces.

```
@cert-authority *.mwl.io,michaelwlucas.com ssh-rsa AAAAB3NzaC1yc2EAAAADAQABA…
```

Add this line to `/etc/ssh/ssh_known_hosts` on each of your hosts. They will

immediately trust certificates signed with this key.

Common Certificate Considerations

Both user and host certificates have a whole bunch of details in common, including serial numbers, certificate IDs, and expiration date format.

Certificate Serial Numbers

Each X.509 certificate is supposed to have a *serial number*. These serial numbers are unique to the certificate authority. OpenSSH CAs support the same functionality, but SSH doesn't really need it. Use them if your organization has some separate need for them.

Serial numbers should not increase monotonically—that is, don't issue them in sequential order. Random serial numbers are best. If you decide to use serial numbers in your SSH certificates, you'll need a mechanism to generate unique random numbers.

I'll mention how to list serial numbers when signing keys, but not spend any time on them.

Certificate Identity

Every certificate has a *certificate identity*, a text string used to say what this key is for. I assign identities based on the hostname or username, but if your organization has other need for the identity feel free to use it. You might decide to use it for inventory tags, personnel ID numbers, or anything else needed. In my examples, I use `host_` and the hostname for host certificates, and `user_` plus the username for user certificates.

Whenever a user authenticates with a certificate, the log message includes the certificate identity. Some people use the certificate identity to let everybody log in as `root` but still retain user accountability.

Certificate Archives

Regenerating certificates requires only a command. You have automation to automatically update and distribute certificates and keys across your network. As people find out just how useful certificates are, the number of certificates you have will multiply. Be sure your CA retains a copy of every certificate and the corresponding public key.

If you discover that a private key has been compromised, you'll need to revoke the certificate for that key. It's much easier to revoke the certificate when you have a copy on hand. When I create a certificate I create a copy of the certificate and its corresponding public key, both prepended with the date in ISO 8601 format (numeric year-month-day). This makes it easy to find certificates with a certain date.

When you create a new CA key and obsolete the old one, you can discard

certificates created with that key.

Setting Expiration Date

Expiration dates are assigned with the standard Unix relative date format. You don't assign a specific end date, but rather how far in the future you want the certificate to expire.

The expiration date begins with a plus sign, plus how far in the future you want. Use `w` to indicate weeks, `d` for days, `h` for hours, `m` for minutes, and `s` for seconds. To have a key expire 56 weeks, 5 days, 12 hours, and 13 seconds from now, use `+56w5d12h13s`.

Host Certificates

OpenSSH clients trust server public keys that have been signed by a recognized certificate authority. Deploying this requires creating certificates and installing those certificates on the server.

Creating Host Certificates

Use your certificate authority to sign a server's public host keys. If you already have a system where you keep copies of each server's public key files, consider either placing your certificate authority on that host or moving the functions that need those files to your CA server.

Sign keys with ssh-keygen(8). Yes, there's an ssh-keysign(8) command, but it's for replacing rsh(1) with SSH and disabled by default. Use the `-s` flag to give the filename of your CA key. The `-I` flag defines the certificate identity. Add the `-h` to declare this is a host key certificate. The `-n` identifies the host this certificate is good for. (You can use multiple host names, separating them by commas.) Use `-V` to give the expiration date, then give the filenames of the key files you want to create certificates for.

Here I use my CA host-mwlca-key to create key certificates for the host **`sloth`**. It expires in fifty-six weeks and five days. I sign keys for every public key file in the current directory. (I must give the full path to the CA key, but the path is trimmed here for clarity.) # **ssh-keygen -s host-mwlca-key -I host_sloth -h -n sloth.mwl.io -V +56w5d ssh_host_*pub**

I copied four public keys to this host, so I get a cert for each of them. Each certificate is named after its public key file, with *`-cert`* inserted before the trailing *`.pub`*. The certificate for *`ssh_host_rsa_key.pub`* is *`ssh_host_rsa_key-cert.pub`*, *`ssh_host_ecdsa_key.pub`* gets *`ssh_host_ecdsa-keycert.pub`*, and so on.

Copy all of these certificates to the SSH server's */etc/ssh* directory.

Installing Host Certificates

Once you have your certificates installed on the server, configure them in *`sshd_config`* with the HostCertificate keyword. As I'm easily confused, I put my

HostCertificate keywords right next to the related HostKey keywords.

```
HostKey /etc/ssh/ssh_host_rsa_key HostCertificate /etc/ssh/ssh_host_rsa_key-cert.pub …
HostKey /etc/ssh/ssh_host_ed25519_key HostCertificate /etc/ssh/ssh_host_ed25519_key-
cert.pub Restart sshd. You are now ready to use host certificates!
```

Testing Host Certificates

Once you configure certificates in sshd(8) and have set up your client's *`/etc/ssh/ssh_known_hosts`*, you're ready to try certificate-based host key validation.

Move your *`$HOME/.ssh/known_hosts`* file out of the way, or delete it if you're really, really confident. Now SSH into the server. You should get a logon prompt without being prompted to verify the host key.

If the host certificate doesn't work correctly, add a `-v` or two to your `ssh` command line. Does the client see the certificate? If not, run `sshd` in debugging mode to see if it's loading the certificate, and if not, why not. Does `ssh` see the certificate, but not recognize it? If so, you fouled up your *`ssh_known_hosts`* entry.

Revoking Certificates

Certificates are great, until someone hacks into your server and grabs a signed keypair. The thief could use that signed key to masquerade as the compromised host. This is bad. Fortunately, you can use the RevokedHostKeys *`ssh_config`* keyword to tell clients not to trust a public key.

```
RevokedHostKeys /etc/ssh/revoked-hosts The revoked host keys file contains a list of
public keys, one per line. The client will not accept these public keys, even if they have
an accompanying certificate.
```

Your enterprise needs the ability to update clients' revoked keys file, and must test it regularly. While real-time updates are best, even a logon script is better than nothing.

Viewing Certificates

You can view the contents of a certificate with `ssh-keygen -L`. Use `-f` to give the certificate file.

```
# ssh-keygen -Lf ssh_host_ed25519_key-cert.pub
ssh_host_ed25519_key-cert.pub:
  Type: ssh-ed25519-cert-v01@openssh.com host certificate Public key: ED25519-CERT
  SHA256:nNtyIQidY3MXAEfpWZ0wzkXKQFnCoQhe0CRIldc4EB8
  Signing CA: RSA SHA256:ZQHNMc2TmWlnygGy9+UoOYFK92RdbguzNi+cX4gA414
  Key ID: "sloth"
  Serial: 0
  Valid: from 2017-12-04T11:52:00 to 2017-12-25T11:53:17
  Principals:
    sloth.mwl.io
  Critical Options: (none)
  Extensions: (none)
```

Perhaps the most vital details here are the key ID (sloth, for the hostname) and the validity dates. If you have multiple certificate authorities, you might find the signing CA field useful. The principals field gives the entities this certificate is valid for, one per line. If you've gotten your key files so mixed up that you need to compare the public key field to the keys on your server, start over.

The fields that define critical options and extensions are useful for user certificates.

User Certificates

User certificates are more complex than host certificates, mainly because users are more complicated than hosts. An SSH user certificate allows you to replicate everything in *authorized_keys*, including the restrictions and limitations discussed in Chapter 12, “Automation.” This requires delving more deeply into SSH certificates, however.

A key concept of an SSH certificate is the *principal*. A principal defines what entities this certificate is for. For a host certificate, the principal is the hostname. A user certificate’s principal is usually the username the certificate is for, but it might also contain limitations, restrictions, and other information. A user certificate without a principal can be used to authenticate as any user. You *must* assign a principal to every certificate, unless you truly want a wildcard authentication certificate.

We’ll refer to the principal throughout this section. Early on you can think of it as the username, but as we proceed the meaning will expand.

Creating and Viewing User Certificates

Get the user’s public authentication key, usually *id_rsa.pub*, and copy it to your certificate authority machine. You’ll need it to generate the certificate. The public key is not confidential, so there’s no risk in sending it across the network.

The command to create a user certificate closely resembles creating a host certificate. Use `-s` to give the path to the user CA key. The `-I` flag defines the certificate identity, and `-n` gives the certificate principal. Use `-V` to define the expiration time. The last argument is the public key file to sign. Here I have my user CA sign the public key of one of my users, making it valid for username **djm**. I set the validity period to 52 weeks, or one year, because if this expires before the user submits it for renewal I’ll entirely blame it on him.

```
# ssh-keygen -s user-mwlca-key -I user_djm -n djm -V +52w id_rsa.pub
```

I enter the CA passphrase and get a certificate file, *id_rsa-cert.pub*. If you’ve never done this before, look at the certificate.

```
# ssh-keygen -Lf id_rsa-cert.pub
id_rsa-cert.pub:
  Type: ssh-rsa-cert-v01@openssh.com user certificate Public key: RSA-CERT
  SHA256:CfVbRUF+AaUcOxml6wI5Cf5nvtjzBDH6NcXaGU4…
  Signing CA: RSA SHA256:CKZFZXRgOy1ji8zmUhU0zjJQfNs9gLEqAwSjA8pB4dg Key ID: "user_djm"
  Serial: 0
  Valid: from 2017-12-04T07:31:00 to 2018-12-23T07:32:37
  Principals:
    djm
  Critical Options: (none)
  Extensions:
    permit-X11-forwarding
    permit-agent-forwarding
```

```
permit-port-forwarding
permit-pty
permit-user-rc
```

While the top looks a whole lot like a host certificate, users get different information below. Our principal is **djm**, so this certificate is only good for this user. We have no critical options, but the Extensions list several keywords. These are the SSH permissions granted to this user, as we'll very soon see in "Restricted Certificates."

Return this certificate to the user.

Using User Certificates

Copy the certificate into *$HOME/.ssh*. The user's public authentication key should already be there.

You should have already set the TrustedUserCAKeys keyword, as discussed in "Trusting your Certificate Authority" earlier this chapter. If so, move the user's *authorized_keys* file aside. Have the user SSH into the server. If the server is properly configured, the user should get in without the server having any information about this particular key.

If this is all you want, you're done. But let's look at some harder stuff.

Revoking User Certificates

Generally, you don't revoke user certificates. You revoke the public keys associated with the certificate, using the RevokedKeys *sshd_config* keyword.

```
RevokedKeys /etc/ssh/revoked
```

One reason to rotate your CA is that it holds down the length of your revoked certificates list. You don't want to have certificates from a laptop stolen five years ago still in your revoked certificates file!

If you have a complicated list of revoked keys, investigate Key Revocation Lists (KRLs) in ssh-keygen(8).

Restricted Certificates

Just as you can limit the access of accounts and keys, you have the power to restrict certificates. You can do this with the `-O` flag to `ssh-keygen`. The `-O` flag has a whole list of possible restrictions identical to those for *authorized_keys* in Chapter 12. These options include `no-agent-forwarding`, `no-port-forwarding`, `no-pty`, `no-user-rc`, and `no-x11-forwarding`. All of these `no-` restrictions have a corresponding `permit-` version: `permit-agent-forwarding`, `permit-port-forwarding`, `permit-pty`, `permit-user-rc`, and `permit-x11-forwarding`. Additionally you have the `source-address` restriction that dictates the IP addresses that can authenticate using this certificate. There's also the `clear` restriction that (much like `restrict` in *authorized_keys*) turns off all privileges, allowing you to turn them on selectively with a permit- statement. Finally, `force-command` compels the user to run that specific command.

One common case for automation is when you have a key that can only run a

single task. I want to create a certificate for a key that can only be used to run the command `/usr/local/scripts/backup.sh`. Create a key called `backup` on the client, and send `backup.pub` to the CA for signing. I want to erase all permissions from this certificate using the clear option, and then compel running the backup script with force-command. I use `-O` twice to assign these permissions. Otherwise, it looks like any other user key signing.

```
# ssh-keygen -s user-mwlca-key -I user_backup -n backup -V +52w -O clear -O force-
command="/usr/local/scripts/backup.sh" backup.pub
```

This generates the certificate file `backup-cert.pub`. Look at the contents.

```
# ssh-keygen -Lf backup-cert.pub
backup-cert.pub:
  Type: ssh-rsa-cert-v01@openssh.com user certificate Public key: RSA-CERT
  SHA256:UL4ctioc5p8aSN1S318FI5RpsS1rnxJdrOEDb/B69Jg Signing CA: RSA
  SHA256:CKZFZXRgOy1ji8zmUhU0zjJQfNs9gLEqAwSjA8pB4dg Key ID: "user_backup"
  Serial: 0
  Valid: from 2017-12-05T07:56:00 to 2018-12-04T07:57:50
  Principals:
    backup
  Critical Options:
    force-command /usr/local/scripts/backup.sh Extensions: (none)
```

Compare this key to the regular user certificate we just created. The user key has no critical options, while this certificate lists the force-command statement as a critical option. Where the user key has a bunch of privileges under Extension, this key has none. This certificate grants the right to use only the one command.

There is no specific privilege to create an SSH tunnel at this time. If your organization is large enough to need certificates, it should have standards declaring acceptable VPN types.[3]

Disabling authorized_keys

Once you've fully deployed SSH certificates for user authentication, you might decide to disable `authorized_keys` files. That's easily done in `sshd_config`.

```
AuthorizedKeysFile none
```

If you have clients that can't support certificates, however, you'll need to provide a way for those clients to log in. Some organizations require all sysadmins to use Unix-based desktops so they can support certificates. Some large organizations like Facebook disallow SSH from clients except to a central bastion host that holds the user's private keys and certificates.

And speaking of Facebook, let's talk about how they manage SSH.

Massive Scale SSH

Organizations like Google, Facebook, and Amazon have tens of thousands of sysadmins and millions of servers. Imagine the load on their LDAP directory just for managing the accounts, and the number of user groups they have.

And once you've imagined that, forget it.

Facebook's engineering team kindly posted an article on how they use SSH

certificates to allow everyone to log in as **root**, but control which servers people can access, through certificate principals. Do an Internet search on “Facebook SSH certificates” and you’ll get right to it. I won’t dive deep into their system, but here’s an overview.

Organizations without millions of servers and teams use usernames as the principal. A principal doesn’t have to be a user, however. You can use the AuthorizedPrincipals *sshd_config* keywords to set up a list of principals that can access the host, and develop principals based on role, location, or function.

The AuthorizedPrincipalsFile keyword points to a text file that contains a list of principals, one per line. Here are three principals that might appear in such a system.

```
everywhere-root
europe-root
europe-database
```

This tells `sshd` to accept authentication from a certificate that includes any of the principals **root-everywhere**, **europe-root**, or **europe-database**. The AuthorizedPrincipalsFile keyword accepts the usual tokens, so you could break this out by username.

```
AuthorizedPrincipalsFile /etc/ssh/principals/%u When a user tries to log in as root, sshd
checks /etc/ssh/principals/root for the list of permitted principals.
```

Assign the principals when you create the certificate. This certificate, for user **mwl**, assigns this certificate the principals **peasants** and **vermin**. As there are many many sysadmins, some with identical names, I store the employee number as well as the name in the key identity.

```
# ssh-keygen -s user-mwlca-key -I user_87181_Michael_Lucas -n peasants,vermin -V +52w
id_rsa.pub
```

This works because the key identity gets logged whenever the key is used to authenticate. Using principals in this way accommodates the need for accountability.

If you’re using this many servers, though, having text files on each server dictating who can log into which account scales badly. You can use the AuthorizedPrincipalsCommand and AuthorizedPrincipalsCommandUser keywords to run a command that fetches the list of authorized principals for this account. This lets your global enterprise with millions of servers continue using that Microsoft Access database for account information—or, yes, you could use LDAP or a modern database like Postgres, if you wanted to be fancy about it.

CA Key Rotation

You can achieve another level of certificate security by rotating your OpenSSH certificate authority keys. This involves creating a new CA keypair, recreating all certificates, distributing those certificates to hosts and users, and removing the old CA’s keys.

It’s possible to deploy SSH certificates without automation—painful, but possible. It’s not impossible to rotate your certificate authorities without automation, but it’s much easier to deploy Ansible to automate the process. Don’t even try to rotate your CA key without automation.

Start by generating your new CA keys and distributing the public keys to each of your hosts. The files that contain trusted CA keys, either `/etc/ssh/ssh_known_hosts` or the file given by the TrustedUserCAKeys `sshd_config` keyword, can contain multiple CA keys simultaneously. Don’t delete the old CA keys yet; only add the new keys.

Once all of your hosts have the public keys for your new CAs, regenerate certificates for all of your hosts and/or users. Distribute those certificates as needed, removing the certificates created with the old CA. Your automation system will report which hosts have the new file and which don’t.

Once your new certificates are distributed, disable the old CA public keys on all of your hosts.

Not only will automation simplify making a key rotation possible, automation makes it possible to rotate your certificate authority keys frequently. If you have a team of sysadmins, forget certificates with a one-year expiration; try one-week host certificates that you update every night! Even if an intruder manages to steal a certificate and a public key, there’s no way they’ll brute-force the private key before the certificate expires.

This is the basics of certificates. Certificates have many small features that can be helpful, if you have the right environment; learn more in ssh-keygen(8). Next, the final chapter takes us through some OpenSSH scraps.

---

[1] SSL is no longer a thing, unless you like bystanders decrypting your traffic.

[2] If you put your SSH CA in `/etc/ssh`, the Sysadmin Code declares that your co-workers are allowed to beat you with a spiked club, provided the spikes are no longer than four inches and not coated with neurotoxin. Local law may vary.

[3] Also, nobody’s asked the OpenSSH maintainer for the feature.

Chapter 15: OpenSSH Scraps

This chapter covers a potpourri of SSH topics that you should probably know about, but that don’t merit their own chapters. We’ll discuss host key rotation in OpenSSH, connecting to hosts that only support obsolete ciphers, and escape characters.

Host Key Rotation

After a host has been accepting connections from the public Internet for a year or two, you should consider rotating the host keys. Not only do algorithms grow easier to break as computing power advances, but prospective intruders have had more time to brute-force your private key. If you ask your users to verify new host keys every year or so, though, they’ll get annoyed. You can use the existing host key to securely transmit the new host key to the client. This isn’t useful if the existing host key has been compromised, but it can let you proactively distribute the forthcoming host keys to clients before getting rid of the old ones.

Once you have many servers, OpenSSH certificates are more useful than occasional key rotation. Certificates eliminate *`known_hosts`* and the need to update the client at all.

Configure SSH key rotation on both the server and the client.

Server Key Rotation

Start by creating your next set of keys. Create each sort of key you intend to support. They’ll need different file names, of course. I name new keys prepended with the year they’re created.

```
# ssh-keygen -f 2018_ssh_host_rsa_key -t rsa -N ''
# ssh-keygen -f 2018_ssh_host_ecdsa_key -t ecdsa -N ''
# ssh-keygen -f 2018_ssh_host_ed25519_key -t ed25519 -N ''
```

This gives us four new keys. Now use the standard *`sshd_config`* HostKey keyword to add these keys. Add new keys after the existing host keys.

```
HostKey /etc/ssh/2018_ssh_host_rsa_key HostKey /etc/ssh/2018_ssh_host_dsa_key HostKey
/etc/ssh/2018_ssh_host_ecdsa_key HostKey /etc/ssh/2018_ssh_host_ed25519_key Your server is
now ready to distribute those host keys to clients.
```

Once you’re certain all of your clients have copies of the new host keys, and you’ve given up waiting for that one user who never updates everything, you can disable the old host keys.

Client Key Rotation

Tell ssh(1) to look for additional keys with the UpdateHostKeys *`ssh_config`* option. The default, `no`, tells `ssh` to ignore new host keys. Setting it to `yes` automatically updates *`known_hosts`* with any new keys for this host. The `ask` setting means to query the user to see if the new keys should be accepted. This mirrors the StrictHostKeyChecking keyword.

When you connect to a host with the UpdateHostKeys option set, your initial connection will look a little different.

```
$ ssh avarice
The authenticity of host 'avarice.mwl.io (203.0.113.209)' can't be established.
ECDSA key fingerprint is SHA256:JUf1lzyEVYxhbJCfXLvPi6eLJdYCZhEBzJD8c+NGLzw.
No matching host key fingerprint found in DNS.
Are you sure you want to continue connecting (yes/no)? yes
```

Verify the host's public key fingerprint and accept it if correct. But then you'll get another set of warnings.

```
Learned new hostkey: RSA SHA256:nNUnWCojrzeHAALXyM/yGpGM7uUIPrP/ph8zV3qUx9M
Learned new hostkey: ED25519 SHA256:nNtyIQidY3MXAEfpWZ0wzkXKQFnCoQhe0CRIldc4…
Accept updated hostkeys? (yes/no): yes
```

Your client has grabbed the public keys for this host's RSA and ED25519 keys. You can only accept or reject these additional public keys en masse. It's a very rare attacker that will leave the main host key untouched while subverting the other keys, but you really should verify them all.

When the server adds new host keys, the client will display the fingerprints and give you a chance to verify them.

```
$ ssh avarice
Learned new hostkey: RSA SHA256:aDqGAPMnT6b3aYqT3DXjRofYfHznOMbVFWZg3yw/fTI Learned new
hostkey: ECDSA SHA256:9eHjmXAFrGmRT2iz/WY5pHLcvoo0HQ5paiLcpEcXWns Learned new hostkey:
ED25519 SHA256:BZ5X6sIbfa5AWcQY0RjnMRL9zLX1+som5TmTV/k/…
Accept updated hostkeys? (yes/no): Host key updates are incompatible with connection
multiplexing (the ControlPersist) keyword. Enabling ControlPersist disables host key
updates.
```

While PuTTY can grab the public key of algorithms it isn't using for a connection (go to the upper left corner menu and select *Special Command -> Cache New Host Key Type*), it can't accept multiple keys of the same type. When you get rid of the old host keys, your PuTTY users must re-verify host keys.

Connecting to Obsolete SSH Servers

Over the last few years, OpenSSH has deprecated a whole bunch of protocols and cryptographic algorithms. The blatantly insecure SSH version 1 has been extirpated from the source code. But whole slews of cryptographic algorithms that worked well in the 1990s are no longer suited to today's Internet. OpenSSH still supports these types of encryption, but they're not enabled by default. You must use special command-line options to use them.

Why disable these algorithms? Awareness. You should know when an SSH connection uses weak crypto. If you never realize that a server or embedded device only supports cruddy cryptographic algorithms, you'll never upgrade or replace it.

When OpenSSH fails to connect to an SSH server due to its weak crypto, it tells you all the information you need to manually connect. You have to understand SSH's encryption characteristics, though.

SSH Encryption

The SSH protocol uses cryptography in four different roles. Each role needs different algorithms. OpenSSH uses a keyword to set each of these in *ssh_config* or on the command line.

The Key Exchange Method (KEX) is used to generate the one-time per-connection symmetric key. The keyword KexAlgorithms sets the key exchange methods.

The general encryption algorithms are set with the Ciphers keyword.

Message Authentication Codes (MAC) detect alterations in traffic. The MACs keyword sets them.

The HostKeyAlgorithms lets you set algorithms for host keys.

Finally, some public key algorithms are obsoleted. The PubkeyAcceptedKeyTypes keyword lets you enable obsolete key types.

Example Connection

My home entertainment network connects to the Internet with an inexpensive embedded router. It offers SSH… sort of.

```
$ ssh admin@203.0.113.1
Unable to negotiate with 203.0.113.1 port 22: no matching host key type found. Their
offer: ssh-dss The router doesn't offer a type of host key that current OpenSSH accepts by
default. The HostKeyAlgorithms keyword lets you re-enable supported but no longer enabled
host key algorithms. The ssh-dss algorithm (also known as DSA) is very weak and abandoned
in modern SSH, but as this is my home network I'll trust it here. Use the
HostKeyAlgorithms keyword to add it back to the options ssh supports.
$ ssh -o HostKeyAlgorithms=+ssh-dss admin@203.0.113.1
Fssh_ssh_dispatch_run_fatal: Connection to 203.0.113.1 port 22: DH GEX group out of range
What, another error? When connecting to an SSH server that only supports obsolete crypto,
you can expect to need to set a few keywords on the command line. Figuring out which are
the necessary settings is an iterative process.
```

This error is a little more obscure. There's no obvious keyword to choose here, unlike with the host key algorithm error. If you're not familiar with Diffie-Hellman key exchange, your best bet is to use an Internet search engine to see if someone's had the same error before. If you're the first person in the entire world to experience this exact problem, run `ssh` in verbose mode, gather the output, and contact the vendor.

This particular error turns out to be a key exchange problem, well-known with this vendor. I have to reactivate an obsolete key exchange algorithm.

```
$ ssh -o HostKeyAlgorithms=+ssh-dss -o KexAlgorithms=diffie-hellman-group14-sha1
admin@203.0.113.1
```

I can now connect.

One day, OpenSSH will fully deprecate these algorithms. Upgrade your equipment before then. As a temporary fix, though, you can set these options in *ssh_config*.

```
Host 203.0.113.1
  HostKeyAlgorithms +ssh-dss KexAlgorithms +diffie-hellman-group14-sha1
```

Now that I've written this section, though, I can upgrade my router.

Escape Characters

When you SSH into a server, your keystrokes all get passed through to the server. With *escape characters*, though, you can talk to the locally running SSH process. An escape character temporarily and briefly suspends your SSH session. You can use the escape character to interrupt a hung SSH session, add port forwarding, send an old-fashioned serial-style break to network gear, and more.

The default escape character is the tilde (~). Very few Unix commands use the tilde, but you can hit it twice to send it once. Hitting `~~` means "yes, I really meant to send a tilde." If you need to change the escape character use `ssh`'s `-e` argument and your desired escape character in quotes.

Issue instructions by hitting `ENTER`, the escape character, and a second character. Disconnecting is `~.`, editing your port forwarding is `~C`, and so on.

Ending Your Session

The easiest use of the escape character is to terminate an SSH session. If the remote server is hung, enter tilde-period.

```
wrath# ~.
Connection to server wrath.mwl.io closed client$
```

You're now back on the local machine.

Adjusting Port Forwarding

The escape character kind of lets you travel backwards in time, adjusting the command you used to connect to the host. While you can't muck with key exchange algorithms and such, you can adjust port forwarding. Enter `~C` to enter the command line, and then enter the desired port forwarding.

Suppose I'm in the middle of an SSH session, and I want to add a dynamic port forward from port 9999 on my desktop out to the server. If I was opening the SSH session with this, I'd add the flag `-D 9999` to the command line. I start by typing `~C`, and get an `ssh>` prompt.

```
ssh>
```

This is the internal ssh(1) command prompt. Add your command line changes here.

```
ssh> -D9999
Forwarding port.
```

Going back to my client, I'll see that ssh(1) has port 9999 open. The dynamic forwarding is live.

To cancel a port forwarding, go back to the SSH command line. Use the `-K` flag and the command you used to create the port forwarding. Here I disable the dynamic forwarding I just created.

```
ssh> -KD9999
Canceled forwarding.
```

The dynamic forward disappears.

Escape characters have other features, but most of them aren't useful today. If

you’re curious, though, ~? displays a list of all available escape characters.

There’s a lot more you can do with SSH. If you can do all of this, you’re more competent with SSH than almost everyone. Congratulations!

Afterword

Seven years ago, I had a temper tantrum about sysadmins managing critical, public-facing systems with password-based SSH.

This isn't anything new. Millions of sysadmins more senior than I have given that rant. I decided to write the first edition of *SSH Mastery* with the explicit purpose of killing passwords. I'm not sure if it helped, but a whole bunch of senior sysadmins have come up to me and thanked me for writing the book, specifically because slapping people with it was considered "professional behavior." I'm hopeful that this second edition, by covering features like certificates, will help those same sysadmins further secure their servers.

Unix users should already know that OpenSSH is one of the most important pieces of security software in the world. If you don't: OpenSSH is one of the most important pieces of security software in the world. Almost every technology vendor includes OpenSSH in their product. These multi-billion-dollar firms don't pay for OpenSSH. Some OpenSSH developers hold a specific day job because their employer gives them time to work on OpenSSH, and companies like Google, Microsoft, and Facebook have donated funds to support the project. For the most part, OpenSSH is created by a bunch of people who love good software.

Running a major software project isn't cheap. OpenSSH is developed as part of the OpenBSD Project. They need servers, bandwidth, and electricity like any other IT organization, but must constantly scrape up funding. If you find OpenSSH useful, consider sending the OpenBSD Foundation (http://www.openbsdfoundation.org/) a few dollars so they can keep going.

Windows folks, PuTTY has revolutionized using SSH from Microsoft systems. And the PuTTY developers gratefully accept donations. They don't have a server infrastructure to feed, but they appreciate donations just the same. With refreshing honesty, they declare that they'll spend small donations on motivational beer and curry, while larger donations can help buy any necessary hardware or tools. Volunteer programmers might have more powerful motivators than Unexpected Appreciation Beer, but I've yet to see what that would be. See the PuTTY FAQ for their PayPal address.

If you work for one of those big firms that make cash out of shipping OpenSSH or PuTTY with their product, do consider ~~blackmailing~~ ~~extorting~~ persuading your employer to throw a few bucks to the folks who write the software. Or at least buy some developers a few pints on your expense account. We'll all benefit.

And if you’re still using passwords after reading this far? I have a whole horde of sysadmins queued up to slap you with a book.

About the Author

Sign up for Michael W Lucas' mailing list.
https://mwl.io

More Tech Books from Michael W Lucas Absolute BSD

Absolute OpenBSD (1st and 2nd edition) Cisco Routers for the Desperate (1st and 2nd edition) PGP and GPG
Absolute FreeBSD
Network Flow Analysis Absolute FreeBSD 3rd edition (coming 2018)

the IT Mastery Series SSH Mastery (1st and 2nd edition) DNSSEC Mastery Sudo Mastery FreeBSD Mastery: Storage Essentials Networking for Systems Administrators Tarsnap Mastery FreeBSD Mastery: ZFS

FreeBSD Mastery: Specialty Filesystems FreeBSD Mastery: Advanced ZFS
PAM Mastery Relayd and Httpd Mastery

Novels (as Michael Warren Lucas) git commit murder git sync murder (coming 2018) Immortal Clay Kipuka Blues Bones Like Water (coming 2018) Butterfly Stomp Waltz Hydrogen Sleets

Sponsors

Somehow, I'm paying the bills as a full-time writer. The only way I've managed that is because people buy my books. I'm grateful to every one of my readers.

A few people like my books so much that they want to help support me. They send me money for a book *as I'm writing that book*. In exchange, I put their names in the print and/or electronic versions of the book. Ebook sponsors paid at least $25 to have their name in the electronic version of SSH Mastery, 2nd Edition, while print sponsors paid at least $125 to get their name on dead trees.

Everyone who contributed: thank you. While I don't *need* sponsorships, they do give me an invaluable financial cushion. You distinctly and directly improve my life.

Print Sponsors

William Allaire Carlos Cardenas Jake Cross Benedict Reuschling Phi Network Systems John W. O'Brien Stefan Johnson Majid Al Suwaidi Mischa Peters Dominique Poulain

Ebook Sponsors

Julien Vallée Alessandro Lenzen Martin Pugh Alexander Riepl Anonymous Jay Nelson Timur Anthony D B

Bernard Spil Roman Zolotarev Steven Hogarth Grant Taylor Matthias Schmidt Danilo Baio Sergio Ligregni John W. O'Brien Mathias Zimmermann Don Jackson Darren Janisse Filipp Lepalaan Viacheslav Bachynskyi Markus Weber Filipe Rodrigues Garance A Drosehn Lucas Holt Aaron Poffenberger Mischa Peters Dominique Poulain Paul Kelly Aubry Hamonic

Patrons

Where the sponsors backed this particular book, a handful of ~~maniacs~~ fine folks sponsor absolutely everything I write, via my Patreon (https://www.patreon.com/mwlucas). The following amazing people send me at least twenty dollars every month.

Digital Supporters

Jeff Marracini
Trent T.
Earl Percival
Allan Jude

Copyright Information