

7

Secure Remote Administration With SSH

7.0 Introduction

In this chapter and the next chapter we'll look at some of the ways Linux offers to remotely administer a server, or to remotely access your workstation. Linux gives users great flexibility and functionality. You may have command-line only or a full graphical desktop, just as though you were physically present at the remote machine.

OpenSSH is the tool of choice for remote command-line administration. It's secure, and easy to set up and use. It's also good for running a remote graphical desktop, because you can tunnel X Windows securely over SSH. This works well over fast local links. However, it's less satisfactory over a dialup or Internet connection because you'll experience significant lag.

Rdesktop is a simple Linux client for connecting to Windows Terminal Servers, and to the Windows XP Professional Remote Desktop. This is useful for some system administration tasks, and for accessing Windows applications from Linux.

For dialup users who want a remote graphical desktop over dialup, FreeNX is just the ticket. It is designed to deliver good performance over slow links. Currently you can use it to access a Linux PC from Linux, Windows, MacOSX, and Solaris.

VNC is the reigning champion of cross-platform remote graphical desktops. With VNC you may do all sorts of neat things: run several PCs from a single keyboard, mouse, and monitor, mix and match operating systems, and do remote technical support.

In this chapter we'll look at how to use OpenSSH. The next chapter is devoted to Rdesktop, FreeNX, and VNC.

OpenSSH

OpenSSH is the Free Software implementation of the SSH protocol, licensed under a modified BSD license that pretty much lets you do whatever you want with it, including modify and re-distribute it, as long as you include the copyright notices.

OpenSSH is used to foil eavesdropping and spoofing on network traffic by

encrypting all traffic during a session, both logins and data transfer. It performs three tasks: authentication, encryption, and it guarantees the integrity of the data transfer. If something happens to alter your packets, SSH will tell you.

There are two incompatible SSH protocols: SSH-1 and SSH-2. OpenSSH supports both of them, but I do not recommend using SSH-1 at all. If you have to log in to remote systems under someone else's control that are still using SSH-1, consider exercising some tough love and telling them you are not willing to risk your security any more, and they must upgrade. SSH-1 was great in its day, but that was then. It has a number of flaws that are fixed by upgrading to SSH-2. See "CA-2001-35" (<http://www.cert.org/advisories/CA-2001-35.html>) for more information, and don't forget to review the list of references at the end of the article

SSH Tunneling

You may use SSH port forwarding, also called "tunneling", to securely encapsulate non-secure protocols like wireless and VNC, which you'll see in various recipes in this book.

OpenSSH supports a number of strong encryption algorithms: 3DES, Blowfish, AES and *arcfour*. These are unencumbered by patents; in fact, the OpenSSH team has gone to great lengths to ensure that no patented or otherwise encumbered code is inside OpenSSH.

OpenSSH Components

OpenSSH is a suite of remote transfer utilities:

`sshd`

The OpenSSH server daemon

`ssh`

Stands for secure shell, though it doesn't really include a shell, but provides a secure channel to the command shell on the remote system

`scp`

Secure copy- this provides encrypted file transfer

`sftp`

Secure file transfer protocol

`ssh-copy-id`

Nice little program for installing your personal identity key to a remote machine's *authorized_keys* file

`ssh-keyscan`

Finds and collects public host keys on a network, saving you the trouble of hunting them down manually

`ssh-keygen`

Generates and manages RSA and DSA authentication keys

`ssh-add`

Add RSA or DSA identities to the authentication agent, `ssh-agent`

`ssh-agent`

Remembers your passphrases over multiple SSH logins for automatic authentication. `ssh-agent` binds to a single login session, so logging out, opening another terminal, or rebooting means starting over. A better utility for this is `keychain`, which remembers your passphrases for as long you don't reboot

Using OpenSSH

OpenSSH is very flexible, and supports different types of authentication:

Host-key Authentication

This uses your Linux login and password to authenticate, and your SSH keys encrypt the session. This is the simplest, since all you need are host keys. An SSH host key assures you that the machine you are logging into is who it claims to be

Public-key authentication

Instead of using your system login, authenticate with an SSH identity key. Identity keys authenticate individual users, unlike host keys which authenticate servers. It's a bit more work to set up, because you need to create and distribute identity keys in addition to host keys. This is a slick way to login into multiple hosts with the same login, plus it protects your system login, because the identity key has its own passphrase. Simply distribute copies of your public key to every host that you want to access, and always protect your private key- never share it

Passphrase-less Authentication

This works like public-key authentication, except that the key pair is created without a passphrase. This is useful for automated services, like scripts and cron jobs. Because anyone who succeeds in thieving the private key can then easily gain access, you need to be very protective of the private key

Using a passphrase-less key carries a bit more risk, because then anyone who obtains your private key can masquerade as you. One way to use passphrases with automated processes is to use `ssh-agent` or the `keychain` utility. These remember your passphrases and authenticate automatically. Their one weakness is they do not survive a reboot, so every time you reboot you have to re-enter all of your passphrases. See Chapter 17, "Remote Access", of the Linux Cookbook, for recipes on how to use these excellent utilities.

Key Types

There are two different uses for authentication keys: host keys, which authenticate computers, and identity keys, which authenticate users. The keys themselves are the same type of key, either RSA or DSA. Each key has two parts: the private and the public. The server keeps the private key, and the client uses the public key. Transmissions are encrypted with the public key, and decrypted with the private key. This is a brilliantly simple and easy-to-use scheme - you can safely distribute your public keys as much as you want

"Server" and "client" are defined by the direction of the transaction- the server must have the SSH daemon running and listening for connection attempts. Client is anyone logging into this machine.

7.1 Starting and Stopping OpenSSH

Problem

You installed OpenSSH, and you configured it to start or not start at boot, according

to your preference. Now you want to know how to start and stop it manually, and how to get it to re-read its configure file without restarting.

Solution

The answer, as usual, lies in */etc/init.d*.

On Fedora use these commands:

```
|# /etc/init.d/sshd {start|stop|restart|condrestart|reload|status}
```

On Debian systems use these:

```
|# /etc/init.d/ssh {start|stop|reload|force-reload|restart}
```

If you elected to not have the SSH daemon run automatically after installing OpenSSH on Debian, you will need to rename or delete */etc/ssh/sshd_not_to_be_run* before it will start up. Or you can run `dpkg-reconfigure ssh`.

The OpenSSH configuration file, *sshd.conf*, must be present, or OpenSSH will not start.

Discussion

Port 22, the default SSH port, is a popular target for attack. The Internet is infested with automated attack kits that pummel away at random hosts. Check your firewall logs- you'll see all kinds of garbage trying to brute-force port 22. So some admins prefer to start up the SSH daemon only when they know they'll need it. Some run it on a non-standard port, which is configurable in */etc/ssh/ssh_config*, for example:

```
|Port 2022
```

Check */etc/services* to make sure you don't use an already-used port, and make an entry for any non-standard ports you are using. Using a non-standard port does not fool determined portscanners, but it will alleviate the pummeling a lot and lighten the load on your logfiles. A nice tool for heading off these attacks is the DenyHosts utility; see Recipe ["Using DenyHosts to Foil SSH Attacks"].

Red Hat's `condrestart`, or "conditional restart", restarts a service only if it is already running. If it isn't, it fails silently.

The `reload` command tells the service to re-read its configuration file, instead of completely shutting down and starting up again. This is a nice non-disruptive way to activate changes.

If you like commands like `condrestart` that are not included with your distribution, you may copy them from systems that use them and tweak them for your system. Init scripts are just shell scripts, so they are easy to customize.

See Also

Chapter 7 of the Linux Cookbook, "Starting and Stopping Linux"

Recipe ["Using DenyHosts to Foil SSH Attacks"]

7.2 Creating Strong Passphrases

Problem

You know that you will need to create a strong passphrase every time you create an

SSH key, and you want to define a policy for that spells out what a strong passphrase is. So, what makes a strong passphrase?

Solution

Use these guidelines for creating your own policy:

- An SSH passphrase must be at least eight characters long
- It must not be a word in any language. The easy way to handle this is to use a combination of letters, numbers, and mixed cases
- Reversing words does not work- automated dictionary attacks know about this
- A short sentence works well for most folks, like "pnt btt3r l*vz m1 gUmz" (Peanut butter loves my gums)
- Write it down and keep it in a safe place

Discussion

Whoever convinced hordes of howto authors to teach "Don't write down passwords" should be sent to bed without dessert. It doesn't work. If you don't want to believe me, how about a security expert like Bruce Schneier: "Write Down Your Password" (http://www.schneier.com/blog/archives/2005/06/write_down_your.html)

"I recommend that people write their passwords down on a small piece of paper, and keep it with their other valuable small pieces of paper: in their wallet."

Easily-remembered passwords are also easily guessed. Don't underestimate the power and sophistication of automated password-guessers. Difficult-to-remember passwords are also difficult to crack. Rarely-used passwords are going to evaporate from all but the stickiest of memories.

I use a hand-written file kept in a locked filing cabinet, in a cunningly-labeled folder that does not say "Secret Passwords In Here", plus my personal sysadmin notebook that goes with me everywhere. If any thief actually searches hundreds of files and can decode my personal shorthand that tells what each login is for, well, I guess she deserves to succeed at breaking into my stuff!

7.3 Setting Up Host-Keys For Simplest Authentication

Problem

You want to know how to set up OpenSSH to login to a remote host, using the simplest method that it supports.

Solution

Using host-key authentication is the simplest way to set up remote SSH access. You need:

- OpenSSH installed on the machine you want to log into remotely
- The ssh daemon must be running on the remote server, and port 22 not blocked
- SSH client software on the remote client

- A Linux login account on the remote server
- Distribute the public host key to the clients

Your OpenSSH installer should have already created the host keys. If it didn't, see the next recipe.

First, protect your private host key from accidental overwrites:

```
| # chmod 400 /etc/ssh/ssh_host_rsa_key
```

Now the public host key must be distributed to the clients. One way is to log in from the client, and let OpenSSH transfer the key:

```
| foobar@gouda:~$ ssh reggiano
The authenticity of host 'reggiano (192.168.1.10)' can't be established.
RSA key fingerprint is 26:f6:5b:24:49:e6:71:6f:12:76:1c:2b:a5:ee:fe:fe
Are you sure you want to continue connecting (yes/no)?
Warning: Permanently added 'reggiano 192.168.1.10' (RSA) to the list of known hosts.
foobar@reggiano's password:
Linux reggiano 2.6.15 #1 Sun June 10 11:03:21 PDT 2007 i686 GNU/Linux
Debian GNU/Linux
Last login: S Sun June 10 03:11:49 PDT 2007 from :0.0
foobar@reggiano:~$
```

Now Foobar can work on Reggiano just as if he were physically sitting at the machine, and all traffic -including the initial login- is encrypted.

The host key exchange happens only once, the first time you login. You should never be asked again unless the key is replaced with a new one, or you change your personal `~/.ssh/known_hosts` file.

Discussion

The public host key is stored in the `~/.ssh/known_hosts` file on the client PC. This file can contain any number of host keys.

It is a bad idea to log in as root over SSH; it is better to log in as an ordinary user, then `su` or `sudo` as you need after login. You can login as any user that has an account on the remote machine with the `-l` (login) switch:

```
| foobar@gouda:~$ ssh -l deann reggiano
```

Or like this:

```
| foobar@gouda:~$ ssh deann@reggiano
```

Don't get too worked up over "client" and "server"- the server is whatever machine you are logging into, and the client is wherever you are logging in from. The SSH daemon does not need to be running on the client.

There is a small risk that the host key transmission could be intercepted and a forged key substituted, which would allow an attacker access to your systems. You should verify the IP address and public key fingerprint before typing "yes". Primitive methods of verification, like writing down the fingerprint on a piece of paper, or verifying it via telephone, are effective and immune to computer network exploits.

For the extremely cautious, manually copying keys is also an option; see Recipe ["Generating and Copying SSH Keys"]

See Also

Chapter 17 of the Linux Cookbook, "Remote Access"

man 1 ssh

man 1 ssh-keygen

man 8 sshd

7.4 Generating and Copying SSH Keys

Problem

Your OpenSSH installation did not automatically create host keys, or you want to generate new replacement host keys. Additionally, you don't trust the usual automatic transfer of the host's public key, so you want to manually copy host keys to the clients.

Solution

Should you create RSA or DSA keys? Short answer: it doesn't matter. Both are cryptographically strong. The main difference to the end user is RSA keys can be up to 2048 bits in length, while DSA is limited to 1024 bits, so theoretically RSA keys are more future-proof. The default for either type of key is 1024 bits.

This example generates a new key pair, using the default host key name from */etc/ssh/sshd_config*. Never create a passphrase on host keys- just hit the return key when it asks for one:

```
# cd /etc/ssh/
# ssh-keygen -t dsa -f ssh_host_dsa_key
Generating public/private dsa key pair.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /etc/ssh/ssh_host_dsa_key.
Your public key has been saved in /etc/ssh/ssh_host_dsa_key.pub.
The key fingerprint is:
26:f6:5b:24:49:e6:71:6f:12:76:1c:2b:a5:ee:fe:fe root@windbag
```

You may wish to be extra-cautious and copy the public key manually via floppy disk, USB key, or `scp` over an existing OpenSSH connection, to avoid any possible hijacking in transit. You need to modify the key if you're going to copy it manually. Here is the original public host key:

```
ssh-dss AAAAB3NzaC1kc3MAAACBAleIrrq77k20kUAh8u3RYG1p0iZKAxLQZQzxJ8422d
+uPRwvVAARFnriNa joJaB9L7qu5D0PCSNCOuBMOIkkyHu jfXJe jQQnMucgkDm8AhMfO8TPyLZ6pG459M
+bfwbsBybyWav7eGvgkkTFZYDEd7HmQK6+Vkd9SYqWd
+Q9HkGCRAAAAFQCrhZsuvIuZq5ERrnf5usmMPXlQkQAAAIUqi61+T7Aa2UjE40hnO8rSVfFcuHE6BCmm0FMO
oJQbD9xFTztZbDtZcna0db5l
+6AYxtVInHjiYPj76/hYST5o286/28McWBF8+j8Nn/tHVUcWSjOE8EJG8Xh2GRxab6AOjgo/GAQlilqMx1JfC
b0lcljVN8VDDF4XtPzqBPhtQAAAIbn7IOv9oM9dUiDZUNXa8s6UV46N4rqcD+HtgkltxDm
+tRiI68kZsU5weTLnLRdZfv/o2P3S9TF3ncrS0YhgIFdGupI//28gH
+Y4sYvrUSoRYJLiDELGml+2pI06wXjPpUH2Ia jr9TZ9eKWDIE+t2sz6lVqET95SynXq1UbeTsDjQ==
root@windbag
```

Delete the hostname at the end of the file, and prefix the key with the fully-qualified domain name and IP address. Make sure there are no spaces between the FQDN and address, and there is one space after the IP address:

```
windbag.carla.com,192.168.1.10 ssh-dss
AAAAB3NzaC1kc3MAAACBAleIrg77k20kUAh8u3RYG1p0iZKAxLQZQzxJ8422d
+uPRwvVAARFnrINaJoJaB9L7qu5D0PCSNCOuBMOIkkyHujfXJJeJQnMucgkDm8AhMfO8TPyLZ6pG459M
+bfwbsBybyWav7eGvgkkTFZYDEd7HmQK6+Vkd9SYqWd
+Q9HkGCRAAAAFQCrhZsuvIuZq5ERrnf5usmMPXlQkQAAIAUqi61+T7Aa2UjE40hnO8rSVfFcuHE6BCmmOFMO
oJQbd9xFTztZbDtZcna0db5l
+6AYxtVInHjiYPj76/hYST5o286/28McWBF8+j8Nn/tHVUcWSjOE8EJG8Xh2GRxab6AOjgo/GAQLi1qMxLJfC
b0lcljVN8VDDF4XtPzqBPHTQAAAlBn7IOv9oM9dUiDZUNXa8s6UV46N4rqcD+HtgkltxDm
+tRiI68kZsU5weTLnLRdZfv/o2P3S9TF3ncrS0YhgIFdGupI//28gH
+Y4sYvrUSoRYJLiDELGml+2pI06wXjPpUH2Iajr9TZ9eKWDIE+t2sz6lVqET95SynXq1UbeTsDjQ==
```

Starting with AAAAB, the file must be one long unbroken line, so be sure to do this in a proper text editor that does not insert line breaks.

You may also use the hostname, or just the IP address all by itself.

If you manually copy additional host keys into the *known_hosts* file make sure there no empty lines between them.

Discussion

How much of a risk is there in an automatic host key transfer? The risk is small; it's difficult to launch a successful "man-in-the-middle" attack, but not impossible. Verifying the host IP address and public key fingerprint before accepting the host key are simple and effective precautions.

It really depends on how determined an attacker is to penetrate your network. The attacker would first have to intercept your transmission in a way that does not draw attention. Then possibly spoof the IP address (which is easy) and public key fingerprint of your trusted server, which is not so easy to do. Since most users do not bother to verify these, most times it's not even necessary. Then when you type "yes" to accept the key, you get the attacker's host key. To avoid detection the attacker passes on all traffic between you and the trusted server, while capturing and reading everything that passes between you and the trusted server.

How hard is it to hijack Ethernet traffic? On the LAN, it's easy- check out the [arp spoof](#) utility, which is part of the Dsniff suite of network auditing and penetration-testing tools. How trustworthy are your LAN users? Over the Internet, the attacker would have to compromise your DNS, which is possible, but not easy, assuming your DNS is competently managed. Or be in a position of trust and a place to wreak mischief, such as an employee at your ISP.

In short, it's a small risk, and the decision is yours.

See Also

[man 1 ssh-keygen](#)

7.5 Using Public-Key Authentication to Protect System Passwords

Problem

You are a bit nervous about using system account logins over untrusted networks, even though they are encrypted with SSH. Or, you have a number of remote servers to manage, and you would like to use the same login on all of them, but not with system accounts. In fact you would like your remote logins to be de-coupled from

system logins, plus you would like to have fewer logins and passwords to keep track of.

Solution

Give yourself a single login for multiple hosts by using public-key authentication, which is completely separate from local system accounts. Follow these steps:

Install OpenSSH on all participating machines, and set up host keys on all participating machines. (Host keys always come first.)

Then generate a new identity key pair as an ordinary unprivileged user, and store it in your `~/.ssh` directory on your local workstation. Be sure to create a passphrase:

```
$ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/home/carla/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/carla/.ssh/id_rsa.
Your public key has been saved in /home/carla/.ssh/id_rsa.pub.
The key fingerprint is:
38:ec:04:7d:e9:8f:11:6c:4e:1c:d7:8a:91:84:ac:91 carla@windbag
```

Protect your private identity key from accidental overwrites:

```
$ chmod 400 id_rsa
```

Now copy your new public key (*id_rsa.pub*) to all of the remote user accounts you'll be using, into their `~/.ssh/authorized_keys2` files. If this file does not exist, create it. Using the `ssh-copy-id` utility is the secure, easy way:

```
$ ssh-copy-id -i id_rsa.pub danamania@muis.net
```

Discussion

`ssh-copy-id` copies identity keys in the correct format, makes sure that file permissions and ownership are correct, and ensures you do not copy a private key by mistake.

The `authorized_keys2` file may be named something else, like `authorized_keys`, or `freds_keys`, or anything you want; just make sure it agrees with the "AuthorizedKeysFile" line in `/etc/ssh/sshd.conf`.

Always put a passphrase on human-user authentication keys- it's cheap insurance. If someone manages to steal your private key, it won't do them any good without the passphrase.

Using public-key authentication combined with `sudo` is a good way to delegate admin chores to your underlings, while limiting what they can do.

Ordinary users may run SSH, which wise network admins know and have policies to control, because all manner of forbidden services can be tunneled over SSH, thereby foiling your well-crafted firewalls and network monitors.

See Also

[man 1 ssh-copy-id](#)

[man 1 ssh](#)

[man 1 ssh-keygen](#)

man 8 sshd

Recipe 8.21, "Granting Limited Rootly Powers With Sudo," the Linux Cookbook

7.6 Managing Multiple Identity Keys

Problem

You want to use different identity keys for different servers. How do you create keys with different names?

Solution

Use the `-f` flag of the `ssh-keygen` command to give keys unique names:

```
|[carla@windbag:~/.ssh]$ ssh-keygen -t rsa -f id_mailserver
```

Then use the `-i` flag to select the key you want to use when you log in to the remote host:

```
|$ ssh -i id_mailserver bart@192.168.1.11  
|Enter passphrase for key 'id_mailserver':
```

Discussion

You don't have to name your keys "id_" whatever, you can call them anything you want.

See Also

man 1 ssh-copy-id

man 1 ssh

man 1 ssh-keygen

man 8 sshd

7.7 Hardening OpenSSH

Problem

You are concerned about security threats, both from the inside and the outside. You are concerned about brute-force attacks on the root account, and you want to restrict users to prevent mischief, whether accidental or deliberate. What can you do to make sure OpenSSH is as hardened as it can be?

Solution

OpenSSH is pretty tight out of the box. There are some refinements you can make; take a look at the following steps and tweak to suit your needs. First, fine-tune `/etc/sshd_config` with these restrictive directives:

```
|ListenAddress 12.34.56.78  
|PermitRootLogin no  
|Protocol 2  
|AllowUsers carla foobar@bumble.com lori meflin  
|AllowGroups admins
```

You may want the SSH daemon to listen on a different port:

```
|Port 2222
```

Or, you can configure OpenSSH to disallow password logins, and require all users to have identity keys with this line in `/etc/ssh/sshd_config`:

```
|PasswordAuthentication no
```

Finally, configure `iptables` to filter traffic, blocking all but authorized bits. (See Chapter ["Building a Linux Firewall"])

Discussion

Specifying the interfaces that the SSH daemon is to listen to, and denying root logins, are basic, obvious precautions.

Protocol 2 means your server will only allow SSH2 logins, and will reject SSH1. SSH1 is old enough, and has enough weaknesses, that it really isn't worth the risk of using it. SSH2 has been around for several years, so there is no reason to continue using the SSH1 protocol.

AllowUsers denies logins to all but the listed users. You may use just the login names, or restrict them even further by allowing them to login only from certain hosts, like *foober@bumble.com*.

AllowGroups is a quick way to define allowed users by groups. Any groups not named are denied access. These are normal local Linux groups in `/etc/group`.

If you prefer, you may use *DenyHosts* and *DenyGroups*. These work the opposite of the Allow directives- anyone not listed is allowed to login. Do not mix Allow and Deny directives; only use one or the other.

Changing to a non-standard port will foil some of the SSH attacks that only look for port 22. However, determined portscanners will find out which port your SSH daemon is listening to, so don't count on it as a meaningful security measure- it's just a way to keep your logfiles from filling up too quickly.

See Also

`man 1 passwd`

`man 5 sshd_config`

Recipe 17.13, "Setting File Permissions on ssh Files", the Linux Cookbook

7.8 Changing a Passphrase

Problem

You want to change the passphrase on one of your private keys.

Solution

Use the `-p` switch with the `ssh-keygen` command:

```
|$ ssh-keygen -p -f ~/.ssh/id_dsa
|Enter old passphrase:
|Key has comment '/home/pinball/.ssh/id_dsa'
|Enter new passphrase (empty for no passphrase):
|Enter same passphrase again:
```

```
| Your identification has been saved with the new passphrase.
```

Discussion

Passphrases are not recoverable. If you lose a passphrase, your only option is to create a new key with a new passphrase.

See Also

```
man 1 ssh-keygen
```

7.9 Retrieving a Key Fingerprint

Problem

You are sending a public host key or identity key to another user, and you want the user to be able to verify that the key is genuine by confirming the key fingerprint. You didn't write down the fingerprint when the key was created- how do you find out what it is?

Solution

Use the `ssh-keygen` command:

```
| [carla@windbag:~/.ssh]$ ssh-keygen -l  
| Enter file in which the key is (/home/carla/.ssh/id_rsa): id_mailserver  
| 1024 ce:5e:38:ba:fb:ec:e7:80:83:3e:11:1a:6f:b1:97:8b id_mailserver.pub
```

Discussion

This is where old-fashioned methods of communication, like telephone and sneakernet, come in handy. Don't use email, unless you already have encrypted email set up with its own separate encryption and authentication, because anyone savvy enough to perpetrate a man-in-the-middle attack will be more than smart enough to crack your email. Especially since the vast majority of email is still sent in the clear, so it's trivial to sniff it.

See Also

```
man 1 ssh-keygen
```

7.10 Checking Configuration Syntax

Problem

Is there a syntax-checker for `sshd_config`?

Solution

But of course. After making your changes, run this command:

```
| # sshd -t
```

If there are no syntax errors, it exits silently. If it find mistakes, it tells you:

```
| # sshd -t
```

```
|/etc/ssh/sshd_config: line 9: Bad configuration option: Porotocol
|/etc/ssh/sshd_config: terminating, 1 bad configuration options
```

You can do this while the SSH daemon is running, so you can correct your mistakes before issuing a `reload` or `restart` command.

Discussion

The `-t` stands for "test." It does not affect the SSH daemon, it only checks `/etc/sshd_config` for syntax errors, so you can use it anytime.

See Also

`man 5 sshd_config`

`man 8 sshd`

7.11 Using OpenSSH Client Configuration Files For Easier Logins

Problem

You or your users have a collection of different keys for authenticating on different servers and accounts, and different `ssh` command options for each one. Typing all those long command strings is a bit tedious and error-prone- how do you make it easier and better?

Solution

Put individual configuration files for each server in `~/.ssh/`, and select the one you want with the `-F` flag. This example uses the configuration file `mailserver` to set the connection options for the server `jarlsberg`.

```
|[carla@windbag:~/.ssh]$ ssh -F mailserver jarlsberg
```

If you are logging in over the Internet, you'll need the fully-qualified domain name of the server:

```
|[carla@windbag:~/.ssh]$ ssh -F mailserver jarlsberg.carla.net
```

IP addresses work too.

Discussion

Using custom configuration files lets you manage a lot of different logins sanely. For example, `~/.ssh/mailserver` contains these options:

```
|IdentityFile ~/.ssh/id_mailserver
|Port 2222
|User mail_admin
```

It's easier and less error-prone to type `ssh -F mailserver jarlsberg` than `ssh -i id_mailserver -p 2222 -l mail_admin jarlsberg`.

Don't forget to configure your firewall for your alternate SSH ports, and check `/etc/services` to find unused ports.

You may open up as many alternate ports as you want on a single OpenSSH server. Use `netstat` to keep an eye on activities:

```
# netstat -a --tcp -p | grep ssh
tcp6      0      0 *:2222   **     LISTEN   7329/sshd
tcp6      0      0 *:ssh    **     LISTEN   7329/sshd
tcp6      0      0 :::ffff:192.168.1.1:2222 windbag.localdoma:35474
ESTABLISHED7334/sshd: carla
tcp6      0      0 :::ffff:192.168.1.11:ssh windbag.localdoma:56374
ESTABLISHED7352/sshd: carla
```

Remember, */etc/sshd_config* controls the SSH daemon. */etc/ssh_config* contains the global SSH client settings.

You may have any number of different SSH client configuration files in your *~/.ssh/* directory.

The SSH daemon follows this precedence:

- command-line options
- user's configuration file (*\$HOME/.ssh/config*)
- system-wide configuration file (*/etc/ssh/ssh_config*)

User's configuration files will not override global security settings, which is fortunate for your sanity and your security policies.

See Also

`man 1 ssh`

`man 5 ssh_config`

7.12 Tunneling X Windows Securely Over SSH

Problem

OK, all of this command-line stuff is slick and easy, but you still want a nice graphical environment. Maybe you use graphical utilities to manage your headless servers. Maybe you want to access a remote workstation and have access to all of its applications. You know that X Windows has built-in networking abilities, but it sends all traffic in cleartext, which of course is unacceptably insecure, plus it's a pain to set up. What else can you do?

Solution

Tunneling X over SSH is simple and requires no additional software.

First, make sure this line is in */etc/ssh/sshd_config* on the remote machine:

```
|X11Forwarding yes
```

Then connect to the server using the *-X* flag:

```
[carla@windbag:~/.ssh]$ ssh -X stilton
Enter passphrase for key '/home/carla/.ssh/id_rsa':
Linux stilton 2.6.15-26-k7 #1 SMP PREEMPT Sun Jun 3 03:40:32 UTC 2007 i686 GNU/Linux
Last login: Sat June  2 14:55:10 2007
carla@stilton:~$
```

Now you can run any of the X applications installed on the remote PC by starting them from the command line:

```
|carla@stilton:~$ ppracer
```

SSH sets up an X proxy on the SSH server, which you can see with this command:

```
|carla@stilton:~$ echo $DISPLAY  
localhost:10.0
```

Discussion

The X server runs with the offset specified in */etc/sshd.conf*:

```
|X11DisplayOffset 10
```

This needs to be configured to avoid colliding with existing X sessions. Your regular local X session is :0.0.

The remote system only needs to be powered on. You don't need any local users to be logged in, and you don't even need X to be running. X needs to be running only on the client PC.

Starting with version 3.8, OpenSSH introduced the `-Y` option for remote X sessions. Using the `-Y` option treats the remote X client as trusted. The old-fashioned way to do this was to configure *ssh_config* with *ForwardX11Trusted yes*. (The *ForwardX11Trusted* default is no.) Using the `-Y` flag lets you keep the default as no, and to enable trusted X forwarding as you need. Theoretically you could find that some functions don't work on an untrusted client, but I have yet to see any.

The risk of running a remote X session as trusted matters only if the remote machine has been compromised, and an attacker knows how to sniff your input operations, like keystrokes, mouse movements, and copy-and-paste. Also, anyone sitting at the remote machine can do the same thing. Oldtimers from the pre-SSH days like to reminisce about their fun days of messing with other user's X sessions and causing mischief.

It is possible to tunnel an entire X session over SSH, and run your favorite desktop or windows manager, like Gnome, KDE, IceWM, and so forth. However, I don't recommend it, because there are easier and better ways to do this, as you will see in the next chapter.

Don't use compression over fast networks, because it will slow down data transfer.

See Also

`man 1 ssh`

`man 5 ssh_config`

7.13 Executing Commands Without Opening a Remote Shell

Problem

You have a single command to run on the remote machine, and you think it would be nice to be able to just run it without logging in and opening a remote shell, running the command, and then logging out. After all, is it not true that laziness is a virtue for network admins?

Solution

And you shall have what you want, because OpenSSH can do this. This example shows how to restart Postfix:

```
|$ ssh mailadmin@limberger.alrac.net sudo /etc/init.d/postfix restart
```

This shows how to open a quick game of Kpoker, which requires X Windows:

```
|$ ssh -X 192.168.1.10 /usr/games/kpoker
```

You'll be asked for a password, but you'll still save one whole step.

Discussion

You have to use `sudo` when you need root privileges with this command, not `su`, because you can't use `su` without first opening a remote shell. This is also a handy way to script remote commands.

And yes, laziness is a virtue, if it leads to increased efficiency and streamlined methods of getting jobs done.

See Also

`man 1 ssh`

7.14 Using Comments to Label Keys

Problem

You have a lot of SSH keys, and you would like a simple way to identify the public keys after they are transferred to your `known_hosts` and `authorized_keys2` files.

Solution

Use the `comment` option when you create a key to give it a descriptive label:

```
|$ ssh-keygen -t rsa -C "mailserver on jarlsberg"
```

The key looks like this:

```
|ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAIEAoK8bYXg195hp+y1oeMWdwlBKdGkSG8UqrwKpwNU9Sbo
|+uGPPNxU3iAjRaLYTniwnoS0j+Nwj+POU5s9KKBf5hx
|+EJT/8w170KyoySlPgHSQAudODOEwCzNFdIME8nm0vxz1AxS
|+S045RxdXB08j8WMDc92PcMOxIB1wPCIntji0= mailserver on jarlsberg
```

This is helpful when you have a lot of keys in `known_hosts` and `authorized_keys2`, because even though you can give the keys unique names, the keynames are not stored in these files.

Discussion

OpenSSH ignores the `comment` field; it's a convenience for human users.

See Also

`man 1 ssh-keygen`

7.15 Using DenyHosts to Foil SSH Attacks

Problem

The Internet is full of twits who have nothing better to do than to release automated SSH attacks on the world. You have taken all the sensible security precautions, and feel like your security measures are adequate, but your logfiles are overflowing with this junk. Isn't there some way to head these morons off at the pass?

Solution

Indeed yes. The excellent DenyHosts utility will take care of you. DenyHosts parses your *auth* log, and writes entries to */etc/hosts.deny* to block future intrusion attempts.

DenyHosts is a Python script, so you need Python 2.3 or newer. Find your Python version this way:

```
$ python -V
Python 2.4.2
```

DenyHosts can be installed with Aptitude or Yum. To install from sources, simply unpack the tarball in the directory where you want to store DenyHosts. This comes with *denyhosts.cfg.dist*, which is a model configuration file. Edit it, then save it as */etc/denyhosts.conf*. (See the next recipe to learn how to configure a startup script.)

Next, create a whitelist in */etc/hosts.allow*; in other words, add all the important hosts that you never want blocked.

This sample configuration is moderately stern. Make sure the filepaths are correct for your system:

```
WORK_DIR = /var/denyhosts/data
SECURE_LOG = /var/log/auth.log
HOSTS_DENY = /etc/hosts.deny
BLOCK_SERVICE = sshd
DENY_THRESHOLD_INVALID = 3
DENY_THRESHOLD_VALID = 5
DENY_THRESHOLD_ROOT = 1
LOCK_FILE = /tmp/denyhosts.lock
HOSTNAME_LOOKUP=NO
SUSPICIOUS_LOGIN_REPORT_ALLOWED_HOSTS=YES
AGE_RESET_VALID=1d
AGE_RESET_ROOT=25d
AGE_RESET_INVALID=
DAEMON_PURGE = 1h
DAEMON_SLEEP = 30s
DAEMON_LOG_TIME_FORMAT = %b %d %H:%M:%S
ADMIN_EMAIL = carla@kielbasa.net
```

The default configuration file tells you the required options, optional settings, and other useful information.

Discussion

DenyHosts can be run manually, as a *cron* job, or as a daemon. I prefer daemon mode- set it and forget it. To run it manually for testing, simply run the DenyHosts script:

```
# python denyhosts.py
```

Read the *denyhosts.py* script to see the available command options.

This is what the options mean:

`BLOCK_SERVICE = sshd`

You may use DenyHosts to protect SSH, or all services with `BLOCK_SERVICE = ALL`

`DENY_THRESHOLD_INVALID = 2`

Login attempts on non-existent accounts get two chances before they are blocked. Since the accounts do not exist, blocking them won't hurt anything

`DENY_THRESHOLD_VALID = 5`

Login attempts on legitimate accounts get five chances. Adjust as needed for fat-fingered users

`DENY_THRESHOLD_ROOT = 1`

Root logins get one chance. You should login as an unprivileged user anyway, then su or sudo if you need rootly powers

`HOSTNAME_LOOKUP = Yes`

DenyHosts will look up hostnames of blocked IP addresses. This can be disabled if it slows things down too much with `HOSTNAME_LOOKUP = NO`

`SUSPICIOUS_LOGIN_REPORT_ALLOWED_HOSTS`

Set this to YES, then monitor your DenyHosts reports to see if this is useful. It tattles about suspicious behavior perpetrated by hosts in `/etc/hosts.allow`, which may or may not be useful

`AGE_RESET_VALID=1d`

Allowed users are unblocked after one day, if they went all fat-fingered and got locked out

`AGE_RESET_INVALID=`

Invalid blocked users are never unblocked

`DAEMON_PURGE = 3d`

Delete all blocked addresses after three days. Your */etc/hosts.deny* file can grow very large, so old entries should be purged periodically

`DAEMON_SLEEP = 5m`

How often should the DenyHosts daemon run? It's a low-stress script, so running it a lot shouldn't affect system performance. Adjust this to suit your situation- if you are getting hammered, you can step up the frequency

Time values look like this:

s: seconds

m: minutes

h: hours

d: days

w: weeks

y: years

See Also

"The DenyHosts FAQ" (<http://denyhosts.sourceforge.net/faq.html>)

7.16 Creating a DenyHosts Startup File

Problem

You installed DenyHosts from the source tarball, so you need to know how to set up an *init* script to start it automatically at boot, and for starting and stopping it manually.

Solution

daemon-control-dist is the model startup file; you'll need to edit it for your particular Linux distribution. Only the first section needs to be edited:

```
#####
# Edit these to suit your configuration #
#####

DENYHOSTS_BIN    = "/usr/bin/denyhosts.py"
DENYHOSTS_LOCK   = "/var/lock/subsys/denyhosts"
DENYHOSTS_CFG    = "/etc/denyhosts.cfg"
```

Make sure the filepaths and filenames are correct for your system. Then give the file a name you can type reasonably, like */etc/init.d/denyhosts*.

Configuring DenyHosts to start at boot is done in the usual manner, using `chkconfig` on Red Hat and Fedora, and `update-rc.d` on Debian:

```
# chkconfig denyhosts --add
# chkconfig denyhosts on

# update-rc.d start 85 2 3 4 5 . stop 30 0 1 6 .
```

Manually stopping and starting DenyHosts is done in the usual manner:

```
# /etc/init.d/denyhosts {start|stop|restart|status|debug}
```

Fedora users also have this option:

```
# /etc/init.d/denyhosts condrestart
```

This restarts DenyHosts only if it already running; otherwise it fails silently.

Discussion

When you create a new *init* script on Fedora, you must first add it to the control of `chkconfig` with the `chkconfig --add` command. Then you can use the `chkconfig foo on/off` command to start or stop it at boot.

See Also

"The DenyHosts FAQ: (<http://denyhosts.sourceforge.net/faq.html>)

Chapter 7 of the Linux Cookbook, "Starting and Stopping Linux"

7.17 Mounting Entire Remote Filesystems With sshfs

Problem

OpenSSH is pretty fast and efficient, and even tunneling X Windows over OpenSSH isn't too laggy. But sometimes you want a faster way to edit a number of remote files- something more convenient than `scp`, and kinder to bandwidth than running a graphical file manager over SSH.

Solution

`sshfs` is just the tool for you. `sshfs` lets you mount an entire remote filesystem and then access it just like a local filesystem.

Install `sshfs`, which should also install `fuse`. Then you need a local directory for your mountpoint:

```
| carla@xena:~$ mkdir /sshfs
```

Then make sure the `fuse` kernel module is loaded:

```
| $ lsmod|grep fuse
fuse                46612  1
```

If it isn't, run `modprobe fuse`.

Next, add yourself to the `fuse` group.

Then log into the remote PC and go to work:

```
| carla@xena:~$ sshfs uberpc: sshfs/
carla@uberpc's password:
carla@xena:~$
```

Now the remote filesystem should be mounted in `~/sshfs` and just as accessible as your local filesystems.

When you're finished, unmount the remote filesystem:

```
| $ fusermount -u sshfs/
```

Discussion

Users who are new to `sshfs` always ask these questions: why not just run X over SSH, or why not just use NFS?

It's faster than running X over SSH, it's a heck of a lot easier to set up than NFS, and a zillion times more secure than NFS, is why.

See Also

`man 1 sshfs`