SSH Tips and Tricks

Ferry Boender

May 2, 2011 (last updated Jan 29, 2019)

Contents

Introduction

SSH is the default unix remote management tool. In its basic form, it is used daily by many administrators as a remote shell in order to issue commands

on different machines. SSH offers much more than just a remote shell though. Mastering it will make your administrator life much easier.

This document provides various usages of SSH besides just as a method of logging in on a remote machine. Some of these tips will be obvious, some less so. In any case, I hope this document will give you some new insights in how SSH can help in your daily operator life.

Authentication

Key-based login

When first starting out using SSH, you'll probably log in with a password. This can quickly become tedious when administering many machines and having to enter your password each time. Instead you can use a public / private key pair to log in.

Generate a new key locally on, e.g. your laptop:

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/user/.ssh/id_rsa): <enter>
Enter passphrase (empty for no passphrase): *************
Enter same passphrase again: *************
Your identification has been saved in /home/user/.ssh/id_rsa.
Your public key has been saved in /home/user/.ssh/id_rsa.pub.
The key fingerprint is:
70:50:84:e3:ea:de:62:43:4e:cf:76:a4:86:8e:c7:29 user@eek
```

This generates two parts of a key: the private part and the public part. You should keep the private key to yourself. We'll upload the public part to a remote server:

```
$ ssh-copy-id -i /home/user/.ssh/id_rsa.pub user@remote_host
```

This puts the public key in a `~/.ssh/authorized_keys` file on the remote host, after which you'll be able to log in using just the key:

```
$ ssh user@remote_host
Enter passphrase for key 'id_rsa':
```

SSH should automatically find the correct key in the `~/.ssh/` directory. If it doesn't you can specify the key manually:

```
$ ssh -i ~/.ssh/some_key.rsa user@remote_host
Enter passphrase for key '.ssh/some_key.rsa':
```

But wait, we still need to enter our password each time? Yes, we need to enter the password that unlocks the key, rather than the password for the remote user.

In the next section we'll look at the SSH Agent, which will keep your key cached in memory, so that you don't need to enter it each time.

SSH Agent

The SSH Agent is a tool which can keep your private keys in memory. When connecting to a remote machine, any SSH session (including `scp` and `sftp`) will try to contact a running agent on the machine to see if the required private key is already loaded. If it is, it will be used to connect to the remote machine. This way you only have to enter your password for a private key once (on your laptop), instead of each time you want to connect.

You can test if an SSH Agent is already running with the following command:

```
$ ssh-add -l
Could not open a connection to your authentication agent.
```

Most modern Linux desktop distributions will already have an agent running. As you can see, in this case no agent is currently running. We can start one with the following command:

```
$ eval `ssh-agent`
Agent pid 6265
```

The above command runs the `ssh-agent` program, which will output some other commands, which are then run by the current shell. This is the actual output of the `ssh-agent` program:

```
SSH_AUTH_SOCK=/tmp/ssh-tXJFfB6269/agent.6269; export SSH_AUTH_SOCK;
SSH_AGENT_PID=6270; export SSH_AGENT_PID;
echo Agent pid 6270;
```

The `ssh-agent` is now running. It creates a socket in the `/tmp` directory and sets that location as an environment variable so that SSH clients (`ssh`, `scp`, `sftp`) know where to contact the agent.

We can add keys to the agent using the `ssh-add` tool:

```
$ ssh-add user.rsa
Enter passphrase for user.rsa:
Identity added: user.rsa (user.rsa)
```

We can now connect to any remote machine that has the public key counterpart in its `authorized_keys` file without having to enter our password again.

The `ssh_agent` program generates a random name for the socket. One frequent problem with starting the SSH agent like we did just now is that only the current shell knows about the socket location. If we open another terminal, it won't know about the running SSH agent, since the environment variable isn't set in the new terminal. We also can't use the `eval` method because it simply starts a

new agent on a different socket. We can work around this by specifying our own path to a socket with the `-a` option.

Combining this knowledge, we can add a few lines to our `.profile` (or `.bashrc`) startup script to start an agent if none is running yet. We also check for a forwarded agent and don't do anything if a forwarded agent is found (more on forwarding agents later on).

```
# Do not start an SSH agent if the user has a forwarded agent.
if [ -z "$SSH_AUTH_SOCK" ]; then
    # Check if a local SSH agent is already running. If not, start one.
    export SSH_AUTH_SOCK="$HOME/.ssh/sshagent.socket"
    if [ ! -S "$SSH_AUTH_SOCK" ]; then
        eval `ssh-agent -a "$SSH_AUTH_SOCK"`
    fi
fi
```

With this script in your `.profile`, you will only have to start one agent ever for a given user as long as that machine doesn't reboot (or the agent is killed in some other way). If you log out, the agent is not killed, and will be reused the next time you log in.

Agent forwarding

One of the most useful feature of SSH is Agent Forwarding. Say we want to log into 'B' in the following scenario:

```
Desktop (with agent)  ->  Machine 'A'  ->  Machine 'B'
```

Agent Forwarding lets us keep our key on our desktop while still being able to login in without a password on Machine 'B', even if we ssh to Machine 'A' first.

Every authentication request made by any SSH session will be sent back to the agent running on your desktop machine, without the need to start additional SSH agents on remote machines and loading keys there. As you can imagine, this is a much better method of keeping private keys secure than storing them on every machine you need to SSH from.

We can enable Agent Forwarding using the `-A` switch to ssh:

```
$ ssh -A user@machine_a
$ ssh -A user@machine_b
```

You can enable SSH agent forwarding for all the hosts you SSH to automatically (without the need to specify the `-A` switch) by putting the following in your `~/.ssh/config`:

```
Host *
    ForwardAgent yes
```

WARNING: Agent forwarding can be dangerous. If an attacker gains access to the socket of an agent on any of the machines you've enabled forwarding for, it will be able to log into any other machine that has your public key. This only works while your agent is active and you are logged into the machine though.

Passwordless key

One very useful feature of SSH are passwordless keys. They are especially useful when you're writing scripts that need to run commands on a remote hosts. Since those will run unattended, you don't want them to prompt for a password.

WARNING: Be careful when using passwordless keys! If the security of the client machine is compromised, the remote server will be *just as compromised*! Check the "Authorization" section of this article for tips on how to limit the damage of a compromised client machine! You should only use passwordless keys for machine-to-machine communication with SSH. For your daily administration tasks, you should use an SSH Agent.

You can generate passwordless keys using the `ssh-keygen` tool. Simply press `enter` when asked for the password:

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/user/.ssh/id_rsa): /home/user/passwordless_rsa
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/user/passwordless_rsa.
Your public key has been saved in /home/user/passwordless_rsa.pub.
The key fingerprint is:
70:50:84:e3:ea:de:62:43:4e:cf:76:a4:86:8e:c7:29 user@eek
```

This generates a normal public key and a private key without a password. We can add the public key to a remote machine like we did in the "Key-based login" chapter.

Now we can ssh to the `remote_host` without using a password. If the private key has been placed in the current user's `.ssh` directory, ssh will automatically detect it when trying to connect. If you want to be sure it finds the private key to connect with, you can once again specify the `-i path_to_private_key` option:

```
$ ssh -i /home/user/passwordless_rsa user@remote_host
user@remote_host$
```

When using passwordless keys, you may provide the `-q` and `-o "BatchMode=yes"` options to SSH, SCP and SFTP in order to make it quiet. This is useful in scripts.

Authorization

Restricting commands

You can restrict which command can be run by someone logging in with a public/private key in the `authorized_keys` file. For instance, to restrict a certain public/private key to running the `df -h` command (to view available disk space), you add a line to the `authorized_keys` file like this (Public key shortened for brevity):

```
command="/bin/df -h" ssh-rsa AAAAC8ghi9ldw== user@host
```

Now when we SSH to that machine (and we have that key loaded):

```
$ ssh user@remote_host
Filesystem            Size  Used Avail Use% Mounted on
/dev/mapper/dev-root   39G  2.2G   35G   6% /
Connection to host closed.
```

Its not possible for a single key to run multiple commands via any normal SSH configuration mechanism. However, SSH will set an environment variable `$SSH_ORIGINAL_COMMAND` with the command the user tried to run. We can take advantage of that by writing a shell script. For example, we can create a script called `commands.sh` on the remote host:

```
#!/bin/sh

case $SSH_ORIGINAL_COMMAND in
    "diskspace")
        df -h
        ;;
    "dirlist")
        ls -1
        ;;
    "apache_restart")
        /etc/init.d/apache restart
        ;;
    *)
        echo "Unknown command"
esac
```

Then we restrict the user to running that shell script:

```
command="/bin/sh /home/user/commands.sh" ssh-rsa AAAAC8ghi9ldw== user@host
```

The user can now run multiple commands by specifying as an argument after the ssh command (The -q‘ option makes ssh quiet, so it doesn’t show any output other than that of the remote command):

```
$ ssh -q user@remote_host diskspace
```

```
Filesystem            Size  Used Avail Use% Mounted on
/dev/mapper/dev-root   39G  2.2G   35G   6% /

$ ssh -q remote_host dirlist
commands.sh
dump.sql
```

WARNING: Make sure you do not include any commands which lets the user run an interactive shell. Also, make sure users cannot overwrite the `commands.sh` and `.ssh/authorized_keys` files.

Restricting allowed IPs

In a similar way as commands, we can restrict from which IPs a login may occur for a specific public key using the "`from`" setting:

```
from="192.168.3.1" ssh-rsa AAAAC8ghi9ldw== user@host
```

We can also specify ranges or extra IPs:

```
from="192.168.3.0/24,10.0.0.1/16" ssh-rsa AAAAC8ghi9ldw== user@host
```

Restricting options

Likewise, we can restrict options:

```
from="192.168.3.1",no-agent-forwarding,no-port-forwarding,no-X11-forwarding ssh-rsa AAAAC8gh
```

Restricting addition of keys

By default, SSH puts the `authorized_keys` files in the user's home directory. This allows users to add other keys themselves; a situation you might want to avoid. You can change the location where the key files are kept in the `/etc/ssh/sshd_config` file, using the `AuthorizedKeysFile` option:

```
#AuthorizedKeysFile     %h/.ssh/authorized_keys
AuthorizedKeysFile      /etc/ssh/authorized_keys/%u
```

The `%u` will expand to the username. The location of the `authorized_keys` file for a user named "john" will become `/etc/ssh/authorized_keys/john`. `%h` expands to the users homedirectory.

Make sure users can't write to the files in question.

Restricting which users can SSH

You can restrict which users are allowed to use SSH with the `AllowUsers` option in `/etc/ssh/sshd_config`:

```
AllowUsers john pete
```

Likewise we can also specify `AllowGroups` to allow all users in that group to log in with SSH.

```
AllowGroups admin
```

NOTE: if the AllowUsers setting is completely missing from the sshd config file, all users are allowed (see man sshd_config). You may prefer to leave it that way – your choice. I prefer to make the usernames explicit because I'm paranoid ;-)

Input / Output

Like every other Unix tool, SSH can use input and output redirection. When running a command on a remote machine using SSH, it will redirect any input given to it locally to the remote command. Any output from the remote command is redirected back to your local machine. This allows for some very useful time-savers.

For instance, we can run the command `du` (diskusage) on the remote machine, and locally pipe it into `xdu` to get a graphical representation on our local X11 desktop of the remote disk usage:

```
$ ssh remote_host du /var/www/ | xdu
```

Or suppose we want to transfer a remote directory's contents to the local machine without using scp (for whatever reason). We can remotely create a tar archive of the directory, and instruct `tar` to write it to the standard output (using the minus as the filename) instead of a file. SSH will transfer the remote standard output of tar to our local machine, where we can untar it in the same manner:

```
$ ssh remote_host tar -cf - Documents/notes | tar -xf -
$ ls Documents/notes/
dev.c.txt                           sysadmin.networking.txt
dev.git.txt                         sysadmin.openssl.txt
dev.mysql.txt                       sysadmin.solaris.txt
```

Perhaps we need to create a local copy of a MySQL database on a remote machine. Unfortunatelly, MySQL access is not remotely allowed and the harddisk on the remote machine is full, so we can't create a dump there, transport it to our local machine and read it in. No worry, SSH to the rescue:

```
$ ssh remote_host mysqldump -u USER -pPASSWORD -h localhost DATABASENAME > dump.sql
```

Or we can just import it directly:

```
$ ssh remote_host mysqldump -u USER -pPASSWORD -h localhost DATABASENAME | mysql -u USER -p
```

Likewise we can locally pipe data into ssh and use it at the remote host. Again, we use the minus-sign to indicate reading from standard in:

```
$ echo "hello world" | ssh remote_host "cat >foo.txt"
```

This will put "hello world" in a file called `foo.txt` on the remote host. We need to quote the parts that contain the redirection on the host (`"cat >foo.txt"`) or it will be picked up by the local shell.

Tunnels and proxies

Local port tunnel

Sometimes you may need to use a certain service on a network, but the network has been firewalled against external connections on ports other than the SSH port. SSH allows us to create a 'tunnel' into the remote network. Suppose we are on a network 192.168.1.x and we want to connect to port 80 on a machine with 192.168.56.3. But the 192.168.56.x network is firewalled, and we can only access it through a bastion host at 192.168.56.1. Here's what we do:

```
$ ssh -L 80:192.168.56.3:80 user@192.168.56.1
```

SSH will now create a tunnel to 192.168.56.3 port 80 through 192.168.56.1. The `-L` option takes three arguments, separated with colons: `local_port:remote_host:remote_port`. The `local_port` is where SSH will listen for incoming connections on the machine where you issued this command. `remote_host:remote_port` is the machine/port to which you wish to create the tunnel. It is important to remember that this is as you'd view it from the server you're ssh-ing too (192.168.56.1 in this case), not as you'd view it from your local machine.

You can additionally specify the `-N` switch to prevent SSH from actually logging in to 192.168.56.1.

SOCKS5 proxy

We can use SSH as a SOCKS5 proxy. An SOCKS5 proxy works much like a normal tunnel, but works with multiple clients at the same time, and is not restricted to forwarding of a single port. We can start a SOCKS5 using the `-D` option:

Socks5 is pretty neat, as it allows you to proxy stuff without the server having to know anything about the way the client works. For instance, if we give the following command:

```
$ ssh -D 8080 remote_host
```

Now we can configure local clients (such as Firefox, Pidgin Instant Messanger, Chrome, etc) to use the proxy. All network traffic (with the exception of DNS, possibly!) will go through the SOCKS5 proxy. For instance:

```
$ chromium-browser --proxy-server="socks5://127.0.0.1:8080"
```

ProxyCommand

Many networks require you to SSH to a bastion (firewall/gateway) server before you're able to SSH to any machine on the network. This becomes tedious quickly, as you have to SSH twice each time. The `ProxyCommand` is a setting in your ssh configuration file which can do this for you automatically.

Assume we want to SSH to a host 'web1.example.com'. Before we can SSH to this host we first have to SSH to 'example.com'. We can SSH directly to 'web1.example.com' by putting the following in our `~.ssh/config` file:

```
Host web1
    ProxyCommand ssh -W %h:%p example.com
```

In older SSH versions, you'd have to use the `nc` (netcat) tool to do this:

```
Host web1
    ProxyCommand ssh example.com nc web1.example.com 22
```

This requires that the `nc` (`netcat`) tool is installed on web1.example.com.

If we SSH to `web1` now, we are automatically sent to web1.example.com. It's even possible to use `scp` and other SSH tricks directly, thus saving us the trouble of having to transfer files to example.com first, then to our local machine (or vice versa).

ProxyCommand can also be used with other things. To SSH through a HTTP proxy at 192.0.2.0 port 8080:

```
ProxyCommand /usr/bin/nc -X connect -x 192.0.2.0:8080 %h %p
```

Client configuration

The SSH client configuration lives in the file `/home/USER/.ssh/config`. It has many useful directives. The basic way it works is we specify a host identifier, and add configuration settings to that host.

For instance, if your local username is `john`, but on the backup machine `backup.example.com` it's always `backup`, you can tell SSH to automatically use that username to log in:

```
Host backup.example.com
    User backup
```

You can create aliases (much like the `/etc/hosts` file) to save some typing:

```
Host backup
    Hostname backup.example.com
    User backup
```

If you want to apply a certain configuration option to *every* host, use the asterisk wildcard:

```
Host *
    ForwardAgent yes
```

To exclude certain hosts, you can use the `!` negation selector:

Host * !untrusted.example.com ForwardAgent yes

Timeouts

When using spotty internet connections or badly configured SSH servers, you may run into disconnects. This is annoying, as you may lose your current work. You can use the following configuration settings to set an extremely long timeout. This will allow your connections to survive even if your internet connection drops for hours.

```
Host *
    # Don't timeout basically ever
    ServerAliveInterval 5
    ServerAliveCountMax 720
    TCPKeepAlive yes
```

One downside to this is that your connection won't timeout even when they should, such as when you've moved from home to the office. It will instead cause your sessions to hang. You can manually disconnect a session by typing `enter`, `~`, and then `.` ("enter", "tilde", "dot"). These should be typed one after another. Don't hold any of the keys at the same time.

ControlMaster

Normally, ssh opens a new connection to a remote host each time you ssh into it. This can be slow, especially if you're jumping through multiple hosts using `ProxyCommand`.

The `ControlMaster` option lets us reuse an already open connection for new connections. This significantly speeds up additional connections to the same host. This is what that looks like:

```
Host *.faraway.com
    ControlMaster auto
    ControlPath ~/.ssh/cm-%r@%h:%p
```

```
ControlPersist 10m
```

Transferring files

Secure Copy (scp)

This should be obvious, but you can use the `scp` tool to transfer files between hosts using SSH. To copy a file `localhost.txt` to your home directory on the remote host:

```
scp localfile.txt user@remote_host:
```

To put the file in a different path:

```
scp localfile.txt user@remote_host:/path/absolute/to/root/
```

and

```
scp localfile.txt user@remote_host:path/in/homedir/
```

Transferring an entire directory is possible using the `-r` switch:

```
scp -r dir/ user@remote_host:
```

SFTP

OpenSSH (and most other SSH implementations) offer a secure FTP server. With it you can securely (encrypted) transfer files and authenticate using the default SSH authentication methods such as passwords or public keys. Contrary to ordinary FTP server (unless they run on TLS or some other form of encryption), passwords are not sent in plain-text over the network. SFTP also makes it easier for Windows user to transfer files between hosts, as there are many good free SFTP clients available. I personally recommend Filezilla.

The SFTP server needs to be enabled in the SSH server configuration. Edit `/etc/ssh/sshd.config` on the server and add the following line:

```
Subsystem sftp /usr/lib/openssh/sftp-server
```

Restart SSH and you should be able to user the SFTP server:

```
/etc/init.d/ssh restart
```

On the client, issue the `sftp` command:

```
$ sftp user@remote_host
Connected to host.
sftp> ls
bin          svn.tar.gz   xims
sftp> get svn.tar.gz
```

```
Fetching /home/user/svn.tar.gz to svn.tar.gz
/home/user/svn.tar.gz           100%   11KB  11.0KB/s   00:00
```

You may get an error like this:

```
subsystem request failed on channel 0
Couldn't read packet: Connection reset by peer
```

This means that either the sftp-server was not properly configurated, or your login shell on the remote server is outputting some text not expected by the SFTP server (perhaps a welcome message or something). Remove the output and try again.

A different way of enabling the SFTP server is in the `authorized_keys` file:

```
command="/usr/lib/openssh/sftp-server" ssh-rsa AAAAC8ghi9ldw== user@host
```

This will restrict any user that logs in using the public key `AAAAC8ghi9ldw==` to SFTP. The user cannot login using a normal SSH session. This does not require you to enable the SFTP server in `/etc/ssh/sshd.conf`.

You'll have to make sure that the user can't write to the .ssh directory nor upload any files such as .bashrc, .profile, etc, otherwise the user can overwrite those by uploading their own version, and they can still execute anything they like by just logging in with sftp. You can do this by creating these files and then changing their ownership and rights in such a way that the user can't write to them. Because it's hard to guess what files you should create so that the user can't cause any harm, it's best to simply create a seperate directory in which they can upload stuff, and lock off write access to their entire home directory.

Unlike the `ssh` and `scp` commands, `sftp` does not have a `-i` switch with which you can manually give the location of the private key to log in with. Fortunately, we can still provide one through the `-o` (options) switch:

```
sftp -o IdentityFile=/home/user/.ssh/some_key_rsa username@hostname
```

This can be convenient in the case of automated tasks. The custom key does not have to have a password and can be placed anywhere. Speaking of automated tasks, here's an example of running `sftp` in a batch-mode:

```
echo "PUT myfile" | sftp -o IdentityFile=/home/user/.ssh/some_key_rsa -b - username@hostname
```

Most normal FTP servers support jailing the user in a certain directory, preventing them from wandering around the file system. SFTP does not have this built-in ability, but we can use normal linux chroot/jails to jail a user to certain directory. For more information, see http://www.electricmonk.nl/log/2007/08/09/jailing-sftpscp/

Remote mount filesystem (SSHfs)

Perhaps the most useful tool in existance: `sshfs`. SSHfs provides tools to locally mount a directory on a remote server over SSH, much like NFS. SSHfs requires remote support for SFTP. See the previous section on how to enable it. SSHfs is separate from the normal ssh tools, and is implemented as a FUSE (userland) filesystem. On Debian and Ubuntu machines you can easily install it using aptitude:

```
# aptitude install sshfs
```

Once installed, we can mount a remote directory using the `sshfs` tool:

```
$ sshfs user@remote_host:path/to/dir ./local_mountpoint
$ cd local_mountpoint
$ ls
file1    file2    file3
```

Unmounting can be done with the `fusermount` tool:

```
$ fusermount -u ./local_mountpoint
```

About this document

Copyright / License