Working Remotely

Overview

Teaching: 10 min
Exercises: 0 min
Questions

- How do I use ‘ `ssh` ’ and ‘ `scp` ’ ?

Objectives

- Learn what SSH is
- Learn what an SSH key is
- Generate your own SSH key pair
- Learn how to use your SSH key
- Learn how to work remotely using `ssh` and `scp`
- Add your SSH key to an remote server

Let’s take a closer look at what happens when we use the shell on a desktop or laptop computer. The first step is to log in so that the operating system knows who we are and what we’re allowed to do. We do this by typing our username and password; the operating system checks those values against its records, and if they match, runs a shell for us.

As we type commands, the 1’s and 0’s that represent the characters we’re typing are sent from the keyboard to the shell. The shell displays those characters on the screen to represent what we type, and then, if what we typed was a command, the shell executes it and displays its output (if any).

What if we want to run some commands on another machine, such as the server in the basement that manages our database of experimental results? To do this, we have to first log in to that machine. We call this a remote login (../reference/#remote-login).

In order for us to be able to login, the remote computer must be running a remote login server (../reference/#remote-login-server) and we will run a client program that can talk to that server. The client program passes our login credentials to the remote login server and, if we are allowed to login, that server then runs a shell for us on the remote computer.

Once our local client is connected to the remote server, everything we type into the client is passed on, by the server, to the shell running on the remote computer. That remote shell runs those commands on our behalf, just as a local shell would, then sends back output, via the server, to our client, for our computer to display.

SSH History

Back in the day, when everyone trusted each other and knew every chip in their computer by its first name, people didn't encrypt anything except the most sensitive information when sending it over a network and the two programs used for running a shell (usually back then, the Bourne Shell, `sh`) on, or copying files to, a remote machine were named `rsh` and `rcp` , respectively. Think (`r`)emote `sh` and `cp`

However, anyone could watch the unencrypted network traffic, which meant that villains could steal usernames and passwords, and use them for all manner of nefarious purposes.

The SSH protocol (../reference/#ssh-protocol) was invented to prevent this (or at least slow it down). It uses several sophisticated, and heavily tested, encryption protocols to ensure that outsiders can't see what's in the messages going back and forth between different computers.

The remote login server which accepts connections from client programs is known as the SSH daemon (../reference/#ssh-daemon), or `sshd` .

The client program we use to login remotely is the secure shell (../reference/#secure-shell), or `ssh` , think (`s`)ecure `sh` .

The `ssh` login client has a companion program called `scp` , think (`s`)ecure `cp` , which allows us to copy files to or from a remote computer using the same kind of encrypted connection.

A remote login using `ssh`

To make a remote login, we issue the command `ssh username@computer` which tries to make a connection to the SSH daemon running on the remote computer we have specified.

After we log in, we can use the remote shell to use the remote computer's files and directories.

Typing `exit` or Control-D terminates the remote shell, and the local client program, and returns us to our previous shell.

In the example below, the remote machine's command prompt is `moon>` instead of just `$` . To make it clearer which machine is doing what, we'll indent the commands sent to the remote machine and their output.

```
$ pwd
```

```
/users/vlad
```

```
$ ssh vlad@moon.euphoric.edu
Password: ********
```

```
    moon> hostname
```

```
    moon
```

```
    moon> pwd
```

```
    /home/vlad
    moon> ls -F
```

```
    bin/     cheese.txt   dark_side/   rocks.cfg
```

```
    moon> exit
```

```
$ pwd
```

```
/users/vlad
```

Copying files to, and from a remote machine using `scp`

To copy a file, we specify the source and destination paths, either of which may include computer names. If we leave out a computer name, `scp` assumes we mean the machine we're running on. For example, this command copies our latest results to the backup server in the basement, printing out its progress as it does so:

```
$ scp results.dat vlad@backupserver:backups/results-2011-11-11.dat
Password: ********
```

```
results.dat              100%   9   1.0 MB/s 00:00
```

Note the colon `:`, seperating the hostname of the server and the pathname of the file we are copying to. It is this character that informs `scp` that the source or target of the copy is on the remote machine and the reason it is needed can be explained as follows:

In the same way that the default directory into which we are placed when running a shell on a remote machine is our home directory on that machine, the default target, for a remote copy, is also the home directory.

This means that

```
$ scp results.dat vlad@backupserver:
```

would copy `results.dat` into our home directory on `backupserver`, however, if we did not have the colon to inform `scp` of the remote machine, we would still have a valid commmad

```
$ scp results.dat vlad@backupserver
```

but now we have merely created a file called `vlad@backupserver` on our local machine, as we would have done with `cp`.

```
$ cp results.dat vlad@backupserver
```

Copying a whole directory betwen remote machines uses the same syntax as the `cp` command: we just use the `-r` option to signal that we want copying to be recursively. For example, this command copies all of our results from the backup server to our laptop:

```
$ scp -r vlad@backupserver:backups ./backups
Password: ********
```

```
results-2011-09-18.dat              100%  7  1.0 MB/s 00:00
results-2011-10-04.dat              100%  9  1.0 MB/s 00:00
results-2011-10-28.dat              100%  8  1.0 MB/s 00:00
results-2011-11-11.dat              100%  9  1.0 MB/s 00:00
```

Running commands on a remote machine using `ssh`

Here's one more thing the `ssh` client program can do for us. Suppose we want to check whether we have already created the file `backups/results-2011-11-12.dat` on the backup server. Instead of logging in and then typing `ls`, we could do this:

```
$ ssh vlad@backupserver "ls results*"
Password: ********
```

```
results-2011-09-18.dat  results-2011-10-28.dat
results-2011-10-04.dat  results-2011-11-11.dat
```

Here, `ssh` takes the argument after our remote username and passes them to the shell on the remote computer. (We have to put quotes around it to make it look like a single argument.) Since those arguments are a legal command, the remote shell runs `ls results` for us and sends the output back to our local shell for display.

SSH Keys

Typing our password over and over again is annoying, especially if the commands we want to run remotely are in a loop. To remove the need to do this, we can create an SSH key (../reference/#ssh-key) to tell the remote machine that it should always trust us.

SSH keys come in pairs, a public key that gets shared with services like GitHub, and a private key that is stored only on your computer. If the keys match, you're granted access.

The cryptography behind SSH keys ensures that no one can reverse engineer your private key from the public one.

The first step in using SSH authorization is to generate your own key pair.

You might already have an SSH key pair on your machine. You can check to see if one exists by moving to your `.ssh` directory and listing the contents.

```
$ cd ~/.ssh
$ ls
```

If you see `id_rsa.pub`, you already have a key pair and don't need to create a new one.

If you don't see `id_rsa.pub`, use the following command to generate a new key pair. Make sure to replace `your@email.com` with your own email address.

```
$ ssh-keygen -t rsa -C "your@email.com"
```

When asked where to save the new key, hit enter to accept the default location.

```
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/username/.ssh/id_rsa):
```

You will then be asked to provide an optional passphrase. This can be used to make your key even more secure, but if what you want is avoiding type your password every time you can skip it by hitting enter twice.

```
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
```

When the key generation is complete, you should see the following confirmation:

```
Your identification has been saved in /Users/username/.ssh/id_rsa.
Your public key has been saved in /Users/username/.ssh/id_rsa.pub.
The key fingerprint is:
01:0f:f4:3b:ca:85:d6:17:a1:7d:f0:68:9d:f0:a2:db your@email.com
The key's randomart image is:
+--[ RSA 2048]----+
|                 |
|                 |
|        . E +    |
|       . o = .   |
|      . S =   o  |
|       o.O . o   |
|       o .+ .    |
|      . o+..     |
|       .+=o      |
+-----------------+
```

The random art image is an alternate way to match keys but we won’t be needing this.

Now you need to place a copy of your public key ony any servers you would like to use SSH to connect to, instead of logging in with a username and passwd.

Display the contents of your new public key file with `cat` :

```
$ cat ~/.ssh/id_rsa.pub
```

```
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEA879BJGYlPTLIuc9/R5MYiN4yc/YiCLcdBpSdzgK9Dt0Bkfe3rSz5cPm4wme
hdE7GkVFXrBJ2YHqPLuM1yx1AUxIebpwlIl9f/aUHOts9eVnVh4NztPy0iSU/Sv0b2ODQQvcy2vYcujlorscl8JjAgfWsO3
W4iGEe6QwBpVomcME8IU35v5VbylM9ORQa6wvZMVrPECBvwItTY8cPWH3MGZiK/74eHbSLKA4PY3gM4GHI450Nie16yggEg
2aTQfWA1rry9JYWEoHS9pJ1dnLqZU3k/8OWgqJrilwSoC5rGjgp93iu0H8T6+mEHGRQe84Nk1y5lESSWIbn6P636Bl3uQ==
your@email.com
```

Copy the contents of the output.

Login to the remote server with your username and password.

```
$ ssh vlad@moon.euphoric.edu
Password: ********
```

Paste the content that you copy at the end of `~/.ssh/authorized_keys` .

```
    moon> nano ~/.ssh/authorized_keys`.
```

After append the content, logout of the remote machine and try login again. If you setup your SSH key correctly you won't need to type your password.

```
    moon> exit
```

```
$ ssh vlad@moon.euphoric.edu
```

SSH Files and Directories

The example of copying our public key to a remote machine, so that it can then be used when we next SSH into that remote machine, assumed that we already had a directory `~/.ssh/` .

Whilst a remote server may support the use of SSH to login, your home directory there may not contain a `.ssh` directory by default.

We have already seen that we can use SSH to run commands on remote machines, so we can ensure that everything is set up as required before we place the copy of our public key on a remote machine.

Walking through this process allows us to highlight some of the typical requirements of the SSH protocol itself, as documented in the man-page for the `ssh` command.

Firstly, we check that we have a `.ssh/` directory on another remote machine, `comet`

```
$ ssh vlad@comet "ls -ld ~/.ssh"
Password: ********
```

```
    ls: cannot access /home/vlad/.ssh: No such file or directory
```

Oh dear! We should create the directory; and check that it's there (Note: two commands, seperated by a semicolon)

```
$ ssh vlad@comet "mkdir ~/.ssh; ls -ld ~/.ssh"
Password: ********
```

```
    drwxr-xr-x 2 vlad vlad 512 Jan 01 09:09 /home/vlad/.ssh
```

Now we have a dot-SSH directory, into which to place SSH-related files but we can see that the default permissions allow anyone to inspect the files within that directory.

For a protocol that is supposed to be secure, this is not considered a good thing and so the recommended permissions are read/write/execute for the user, and not accessible by others.

Let's alter the permissions on the directory

```
$ ssh vlad@comet "chmod 700 ~/.ssh; ls -ld ~/.ssh"
Password: ********
```

```
    drwx------ 2 vlad vlad 512 Jan 01 09:09 /home/vlad/.ssh
```

That's looks much better.

In the above example, it was suggested that we paste the content of our public key at the end of `~/.ssh/authorized_keys` , however as we didn't have a `~/.ssh/` on this remote machine, we can simply copy our public key over as the initial `~/.ssh/authorized_keys` , and of course, we will use `scp` to do this, even though we don't yet have passwordless SSH access set up.

```
$ scp ~/.ssh/id_rsa.pub vlad@comet:.ssh/authorized_keys"
Password: ********
```

Note that the default target for the `scp` command on a remote machine is the home directory, so we have not needed to use the shorthand `~/.ssh/` or even the full path `/home/vlad/.ssh/` to our home directory there.

Checking the permissions of the file we have just created on the remote machine, also serves to indicate that we no longer need to use our password, because we now have what's needed to use SSH without it.

```
$ ssh vlad@comet "ls -l ~/.ssh"
```

```
    -rw-r--r-- 2 vlad vlad 512 Jan 01 09:11 /home/vlad/.ssh/authorized_keys
```

Whilst the authorized keys file is not considered to be highly sensitive, (after all, it contains public keys), we alter the permissions to match the man page's recommendations

```
$ ssh vlad@comet "chmod go-r ~/.ssh/authorized_keys ; ls -l ~/.ssh"
```

```
    -rw------- 2 vlad vlad 512 Jan 01 09:11 /home/vlad/.ssh/authorized_keys
```

Key Points

- SSH is a secure alternative to username/password authorization
- SSH keys are generated in public/private pairs. Your public key can be shared with others. The private keys stays on your machine only.
- The 'ssh' and 'scp' utilities are secure alternatives to logging into, and copying files to/from remote machine

(../01-man-pages/)

(../03-file-transfer/)