SSH KEY AUTHENTICATION

The SSH protocol has support for multiple different types of authentication. By default, one of the authentication mechanisms used by SSH is password authentication. SSH will allow you to connect to a given account on a computer if you know the password for that user on that computer. By default SSH will also try to use an SSH key-pair for authentication instead of a password, and in fact, it will try to use a key-pair before it tries to use a password.

To use SSH key authentication you need to do the following:

1. Generate an SSH key-pair on the computer you will be SSHing from (you only have to do this once, you can use the same key-pair to authenticate to multiple computers safely).
2. Give the **public** key from that key-pair to the person managing the computer you want to SSH to (never share your private key with anyone!).
3. Wait for the administrator of the remote computer to add your public key to the list of allowed keys within the account you will be SSHing to.

Once those steps have been completed you will be able to log in to the remote computer without having to know the password of the user you will be connecting as. Lets look at these steps in detail now. To play along you'll need two computers, one to SSH from, and one to SSH to.

Generating an SSH Key-Pair

The process stars on the computer you will be SSHing from. You need to open a terminal as the user who will be SSHing to the remote computer, and in that terminal type the command:

```
ssh-keygen -t rsa
```

This will create an SSH key-pair and offer to store the two halves in the default locations (press enter to accept the defaults):

- The private key: **~/.ssh/id_rsa**
- The public key: **~/.ssh/id_rsa.pub**

If you already have a set of keys and don’t want to replace them, you can use the**-f** flag to specify a different location to save the private key (the public key will get stored in the same folder and with the same name, but with **.pub** appended to it). When you run the **ssh-keygen** command you will be asked to enter a password. This is the password that will secure the private key. This is a very important safety measure, because it means that if your private key is lost or stolen, it cannot be used unless the attacker also knows the matching password. The **ssh-keygen** command will accept a blank password, but this is to be strongly discouraged because it leaves your private key unprotected.

It should also be noted that if you forget the password protecting your private key, you won't be able to use that key-pair any more, and you'll need to generate a fresh key-pair!

Once you enter a password **ssh-keygen** will generate a public and private key, tell you where it has saved them, tell you the key's fingerprint (a big long hexadecimal string separated with **:**s), and it will show you the key's random art image. This is representation of the key as a little ASCII art graphic. This is much more memorable to humans than the fingerprint, show us two different pictures and we'll spot the difference in seconds, show us two different strings of hex and we'll find it very hard to spot subtle differences!

To get a sense of how difficult an SSH key is to brute-force attack, you can have a look at the private key you just generated with the command:

```
cat ~/.ssh/id_rsa
```

And the public key with the command:

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

Asside – Base64 Encoding

If you are wondering what format the keys are stored in, it's the very commonly used base64 encoding. This is a very robust format that ignores characters like line breaks and spaces which could get introduced if a key were to be copied and *pasted into an email or something like that.*

Granting Access With an SSH Public Key

The next step in the process is to share your public key with the person administering the computer you will be SSHing to. You can do this by attaching the public key to an email, or simply copying and pasting it's content into an email. If we know the password of the remote account we will be connecting to, we can also copy the key over ourselves, but more on that later.

To grant a remote user access to a given account a computer administrator needs to add the remote user's public key to a special file in the home directory of the local user the remote user will be connecting as. That special file is**~/.ssh/authorised_keys** (or, if the key is only to be used over the SSH2 protocol,**~/.ssh/authorized_keys2**).

The **~/.ssh/authorized_keys** file should contain one public key per line. You can grant access to as many users as you like by adding as many public keys are you like.

SSH is an absolute stickler about the permissions on the **authorized_keys** file, including the permissions on the folder that contains it, i.e. **~/.ssh/**. No one other than the owner of the account (and root) should have write permissions to either the containing folder, or the file itself.

Because public keys are not sensitive information, SSH does not care of other users can read what is effectively public information, but the ability to write to that file would allow any other user on the system to grant themselves access to that account by adding their own public key to the list. To prevent this from happening, SSH will not accept a key if it’s contained in a file that is writeable by anyone but the owner of the account. An example of working permissions on an account with the username **bart** is show below:

```
[bart@www ~]$ ls -al ~/.ssh
total 20
drwx------  2 bart bart 4096 May  5  2014 .
drwxr-xr-x 16 bart bart 4096 Mar 15 14:32 ..
-rw-r--r--  1 bart bart  670 Feb 14  2013 authorized_keys
-rw-r--r--  1 bart bart  660 May  5  2014 known_hosts
[bart@www ~]$
```

Remember that in a list of the contents of the folder **~/.ssh**, the permissions on that folder itself are the permissions on the special file **.**. I have highlighted the command, and the two important sets of permissions in bold.

Simplifying the Process with ssh-copy-id

It takes time and effort to manually copy across your public key, and to make sure all the file permissions are correct.

Assuming you know the password to log in to the remote computer, you can automate the process with the **ssh-copy-id** utility.

This utility comes as standard on all the Linux distributions I have used, but annoyingly, OS X's version of SSH does not come with **ssh-copy-id**. All is not lost though, because the open source community are here to help!

OS X users can install **ssh-copy-id** onto their Mac using the free and open source project ssh-copy-id-for-OSX.

If you follow the link above you'll see that installing **ssh-copy-id** onto your mac is as simple as running the command:

```
curl -L https://raw.githubusercontent.com/beautifulcode/ssh-copy-id-for-OSX/master/install.sh | sh
```

The above command has to be run from an admin account, and it uses **sudo** for the install, so you will be prompted for your password.

What ever OS you are on, once you have **ssh-copy-id** installed, copying over your public key becomes as easy as running the command below (replacing **user** and **computer** as appropriate):

```
ssh-copy-id user@computer
```

SSHing to a Computer Using Key Authentication

Once you have generated your key-pair, and the remote admin has correctly added your public key to the **authorized_keys** file, you are ready to start using your private key as your authentication when SSHing to that remote computer.

If you saved your key to the default location (**~/.ssh/id_rsa**), then you don't have to do anything special to start using your key, just issue your SSH command as normal. Remember, by default, SSH tries key-based authentication before password-based authentication. If your private key is not in the default location you need to tell SSH what key to use with the **-i** flag (i for identity).

Assuming you followed best-practice advice and protected your private key with a password, you will be asked for a password when you try to SSH, but you are not being asked for the password of the remote account you are connecting to, instead, you are being asked for the key to unlock your private key.

Securely Saving Your Private Key's Password

I promised convenience AND security, but surely swapping one password for another is no more convenient?

The good news is that there are mechanisms for safely caching that password so you don't have to keep entering it each time you SSH. The exact details of the mechanism vary from OS to OS.

The good news is that Mac users have it best in this regard.

The version of SSH that ships with OS X has support for OS X's secure keychain. This is a secure vault OS X uses to store the passwords you save in all sorts of apps, including Mail.app and Safari. This means that on OS X, when you use SSH key authentication, a popup window will appear asking for the password for your private key, and that pop window has a checkbox to allow the password be saved in your keychain. Once you do this you will never have to enter that password again, you will now be able to SSH without entering a password in a secure manner.

Users of other OSes are not completely out of luck, but the solutions available are less convenient. On Linux and other versions of Unix, a service called **ssh-agent** can be used to cache the passwords for SSH keys. Since this series is targeted primarily at Mac users, I won't go into the details here, but there are plenty of guides available online if you search for *'ssh-agent tutorial'*.

So, whether you are using OS X's key chain, or **ssh-agent**, you can now securely log in to remote computers over SSH with the minimum of effort.

Source: https://www.bartbusschots.ie/s/2015/03/29/taming-the-terminal-part-30-of-n-sshing-more-securely/