ADVANTAGES TO KEY-BASED AUTHENTICATION

- **Convenience** – with OS X's key chain or **ssh-agent** securely storing the password for your private key, you can safely use SSH without having to enter a password.
- **Security** – once you have key-based authentication in place, you can either set a really long and secure password on the remote account, or even disable password-based logons completely (we don't cover how to do that in this series). SSH keys are much more difficult to brute force than even the most complex of passwords.
- **A Form of 2-Factor Auth*** – in order to log in as you, an attackers needs to have your private key, and needs to know the password for your private key. * *Some argue that this is only 1.5 factor auth because unlike a physical dongle, you have no real way of knowing if someone has stolen a copy of your private key – since it is digital, a copy can be taken without depriving you of your copy, and hence alerting you to its loss.*

One place where key-based auth really comes into its own is with shared accounts. Imagine you are working on a website together with some volunteers from a club you are a member of. The server hosting your site allows logins over SSH.

All those working on the project need to be abel to log into the web server to edit the site. Being a club, there is going to be a natural churn of members, so people will continually join and leave the project, and it's possible that some of the leavers will not be leaving on good terms. How do you handle this situation?

First, lets look at the simplest and perhaps most obvious solution – a shared password. You set a password on the account, and share that password with the group. Then, each time a new member starts, you let them in on the secret. So far so good. Then, someone leaves the project. You now have to either accept the fact that someone no longer working on the project still knows the shared secret, and hence can still log in and perhaps sabotage the site, or, you need to change the password and tell only the remaining people the new password. That scheme is workable, but cumbersome.

A better solution would be to give no one the password to the account at all, and use SSH keys instead. On joining the project, each participant provides their SSH public key, and those keys are added to the **~/.ssh/authorized_keys** file. As people come and go, simply add and remove their public keys. When someone leaves, no one else has to change anything, and there is no shared secret.

Managing a long **authorized_keys** file does not have to be difficult for two reasons.

Firstly, **ssh-keygen** adds the username and hostname of the person who's key it is to the end of all public keys, so just reading the key could well tell you all you need to know to identify which key belongs to whom. If that information is not sufficient, you can add comment lines to the file by staring those lines with the # symbol.

Conclusions

Usually we have to choose between convenience and security, but with SSH keys we get to have our proverbial cake and eat it. By putting in a little work up front, we get a more convenient and more secure SSH experience.

So far we have only looked at using SSH to execute terminal commands remotely, either one command at a time, or through an interactive command shell running on the remote computer. But, SSH's encrypted connection can be used to secure much more than just a command shell. In fact, it can be used to secure just about any kind of network communication through a number of different mechanisms. In the next two instalments we'll see how to securely transmit files over SSH, and, how to securely tunnel any network connection through an SSH connection.

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