yubico

# Secure Shell With A YubiKey

White Paper

**September 1, 2015**

**By Alessio Di Mauro**

## Copyright



## Trademarks



## Disclaimer



## Contact Information

**Yubico Inc**
420 Florence Street, Suite 200
Palo Alto, CA 94301
USA
yubi.co/contact

yubico

# Contents

# Introduction

One feature of the YubiKey NEO and NEO-n that many are not aware of is the ability to use the devices together with SSH (Secure Shell) to establish secure connections with remote servers. Let us start the discussion by explaining what SSH is and why it is a good idea to use it.

In ye olde days, when the internet started to become popular and running services on remote computers became a possibility, tools and protocols such as telnet, remote shell (rsh) and the file transfer protocol (FTP) started to appear. While these applications fulfilled their purpose, they were (and still are) regarded as insecure. All of them suffer by sending authentication credentials and commands in plain-text. That is unacceptable today.

To address this problem, secure shell or SSH came into existence in 1995.

## SSH in Short

The main goal of SSH is to provide an authentic and confidential channel over a potentially insecure and untrusted network such as the Internet.

SSH started out as free software but after gaining popularity it began to gravitate towards proprietary implementations. To answer this, the open source community (and more precisely Theo de Raadt, the mind behind OpenBSD) started to work on a free variant called OpenSSH, which saw the light of day in 1999 and has now become the de facto implementation of the SSH protocol. OpenSSH provides secure versions of the tools and protocols described above, in the guise of *ssh*, *scp* and *sftp*. Official ports and re-implementations of the protocol have made SSH available on all major operating systems, and effectively making it the go-to protocol for remote service management.

To achieve its security goals, SSH uses public key cryptography to authenticate clients and servers to one another. Once this is done, a random symmetric session key is generated and used to secure the communications until the connection is torn down. The key is then discarded.

An interesting point is that the protocol provides two major ways of performing authentication. The first is to use a password. This translates into the SSH service generating asymmetric key pairs to perform channel encryption that allows for confidential password transmission. Once received, the password is used to perform the actual authentication.

A second way to perform authentication is through user-generated asymmetric keys. The public key of each authorized user is installed onto the server, and during the authentication phase the owner of the matching private key can prove his identity without having to transmit the key itself.

Key authentication is usually preferred over password authentication and is considered more secure, mainly because passwords come with the usual plethora of issues (not random enough, guessable, crackable, etc). Another equally important but more subtle concern around passwords is that in order to perform an authentication attempt an adversary does not need any external information, just a username and a password, hence brute force attacks are a real possibility unless other countermeasures are taken. Given these points, it's clear that key authentication is a superior method.

# The Best of Both Worlds

It is possible to make things better by protecting the keys with a passphrase. This method is natively supported by SSH and requires little effort. However, as pointed out in this insightful post by Martin Kleppmann, the default procedure used by SSH isn't too great.

SSH uses AES to encrypt the private key and in order to derive the required symmetric key from the user passphrase, a portion of a publicly known value (i.e. the initialization vector) is appended to the passphrase, which is then fed to a single round of MD5, a hashing algorithm. The result is the AES key used to encrypt the private key.

There are two main problems with this approach. First, MD5 is considered insecure nowadays and its use is discouraged. Second, and most important, both MD5 and AES are considerably fast to compute. This means that unless the passphrase is very long and complex, it is susceptible to brute force attacks.

In his post, Kleppmann suggests to convert the key from SSH's own format to PKCS#8, a more sound and standardized method for storing encrypted passphrases. This is a neat solution because OpenSSH relies upon OpenSSL, which transparently supports PKCS#8. With this you can just run a couple of OpenSSL commands and convert your private key format. Unfortunately, this does not work on Mac OS X since version 10.9 (still not working on 10.10) due to a change of libraries.

Besides the OS compatibility problem, there are other issues. When using SSH with asymmetric keys, the private key of the client has to be installed on each and every system from which a user wants to connect. This has both portability and security consequences.

If the key is not available, no connection is possible. If the system used to connect from is public, the private key (although passphrase-protected) could be copied by an attacker. An alternative could be to use a USB thumb-drive to store all your private keys, but this would create a big security concern should you lose or misplace it as there is usually no access protection.

The good news, however, is that there is an easy, elegant, and secure solution to all of these problems: using a YubiKey.

## YubiKey + SSH

The YubiKey NEO (both the keychain and nano form factors) comes preprogrammed with an OpenPGP applet and can store three private subkeys: one for signatures, one for encryption and one for... authentication. BINGO!

Although PGP and SSH are two different protocols, they both support RSA, and after all is said and done, a cryptographic key does not care about protocols or applications. So, how do we convert our authentication key to a format that can be used by OpenSSH? Luckily for us, there is a purpose made application called *gpgkey2ssh* which serves exactly this purpose (in Debian this is provided by the gnupg-agent package). It is not the most brilliant piece of software and the documentation is lackluster to say the least, but provide it with a PGP key-id (make sure it is the key-id of your authentication subkey) and you will be given back the corresponding SSH format.

An alternative to gpgkey2ssh is provided by the `ssh-add` command, but that requires some additional steps (which are required anyway), so if you want to go that route, keep reading.

Let us assume that you were to grant me SSH access to one of your computers, you could retrieve my public key from one of the public key servers, and ideally you would want to validate the key by verifying its fingerprint with me. At this point you can run the command:

```
$ gpgkey2ssh 0A7C787E
```

That command lets you obtain the SSH-compliant version of my public key. All that is left to do now is to instruct the SSH service running on the client system to retrieve private keys from a YubiKey. The component dedicated to managing secrets within SSH is called *ssh-agent*, whereas the component that is able to access private keys within gpg is called, surprise surprise, *gpg-agent*. The idea is to enable communication between gpg-agent and ssh-agent to transmit the private keys. The good news is that gpg-agent already knows how to communicate with ssh-agent, and all that is required is to enable this channel by using the option *enable-ssh-support* in the *gpg-agent.conf* file. By running the command `ssh-add -L` at this point it is possible to see the same output generated by gpgkey2ssh, as a proof that everything is working.

The advantage of this configuration is considerable. The private key is now stored within the secure element of the YubiKey, providing an additional layer of security. It only can be accessed after presenting the correct user PIN. On top of that the actual key is never extracted from the chip. Data will go in as plain-text and come back out in the form of cipher-text/digital signature. Furthermore, we are using only the authentication subkey of our personal keyring, meaning that even if they key does get compromised, the master key is still safe (especially if you store it offline) allowing us to revoke the compromised key and to generate a new one. Last but not least, OpenPGP specifications require the implementation of an attempt counter connected to each subkey stored on a smart card. Its default value is set to three and it is decremented at each failed attempt to access a private key. If the counter reaches zero the related key is blocked. This means that even if you happen to lose your YubiKey, whoever finds it will get three shots at authenticating as yourself before being locked out for good.

Once you have configured your computer to use SSH keys from a YubiKey you are set to use them with your personal server or one of the many services that allow public-key authentication such as GitHub or Bitbucket.

Finally, the cool factor of plugging in your YubiKey, ssh-ing into your home server, being asked for a PIN and then granted access has its appeal, at least to me! Besides the security advantages, the beauty of this approach is that the server does not have to be aware of where the private key is stored, this means that no modifications are required.