July 2021

A Practical Guide to Secure SSH Access

A Teleport Tech Paper

Teleport

Table of Contents

Introduction

There's no denying that Secure Shell Protocol (SSH) is the de facto tool for Linux server administration, but is it truly secure? While SSH is widely supported, fairly straightforward to use, and was built with security in mind, it's far from perfect.

SSH, invented in 1995, is a cryptographic network protocol for accessing UNIX machines securely over an unsecured network. SSH uses a client–server architecture, connecting an SSH client application with an SSH server. It superseded Telnet protocol by adding encryption to prevent malicious actors from eavesdropping on network traffic and compromising data confidentiality and integrity.

Typical applications include remote command and shell execution, but any network endpoint can be secured with SSH. SSH is very important in cloud computing to solve secure connectivity problems to cloud-based virtual machines. Many popular products and just about every server deployment system integrates with SSH somehow. It is universally supported across pretty much all architectures and distributions, from Raspberry Pis all the way up to massive supercomputer clusters.

Since SSH is a powerful tool which often grants a lot of access to anyone using it to login into a server, it's critical to make your SSH model as secure as possible. This tech paper proposes three approaches to achieving that, covering each in a chapter:

- **Chapter 1** - Choosing the right encryption
- **Chapter 2** - Reducing your attack surface area with an SSH proxy
- **Chapter 3** - Using industry best practices

Chapter 1 - Choosing the right encryption

The “secure” in Secure Shell Protocol comes from the combination of hashing, symmetric encryption, and asymmetric encryption (also known as public key encryption). In the 25 years since SSH was created, increasing computer processing power and speeds have necessitated increasingly complicated low-level algorithms.

This chapter focuses on encryption algorithms used by SSH. The most widely adopted algorithms, as of 2020, have been RSA, DSA, ECDSA, and EdDSA. *RSA and EdDSA provide the best security and performance*. To explain why, below is an overview of how encryption happens within SSH, followed by an overview of the four algorithms and the criteria for comparing them.

Encryption within the SSH protocol

SSH is used almost universally to connect to shells on remote machines. The most important part of an SSH session is establishing a secure connection, which occurs via negotiation & connection, and authentication.

Negotiation & connection

For an SSH session to work, both client and server must support the same SSH protocol version.

- After coming to a consensus on which protocol version to follow, both machines negotiate a per-session symmetric key to encrypt the connection from the outside. Generating a symmetric key at this stage, when paired with the asymmetric keys in authentication, prevents the entire session from being compromised if a key is revealed.

- Negotiation terms happen through the Diffie-Helman key exchange, (**KEX**) which creates a shared secret key to secure the whole data stream by combining the private key of one party with the public key of the other. (These keys are different from the SSH keys used for authentication.)

Once the negotiation and connection are complete, a reliable and secure channel between the client and server has been established.

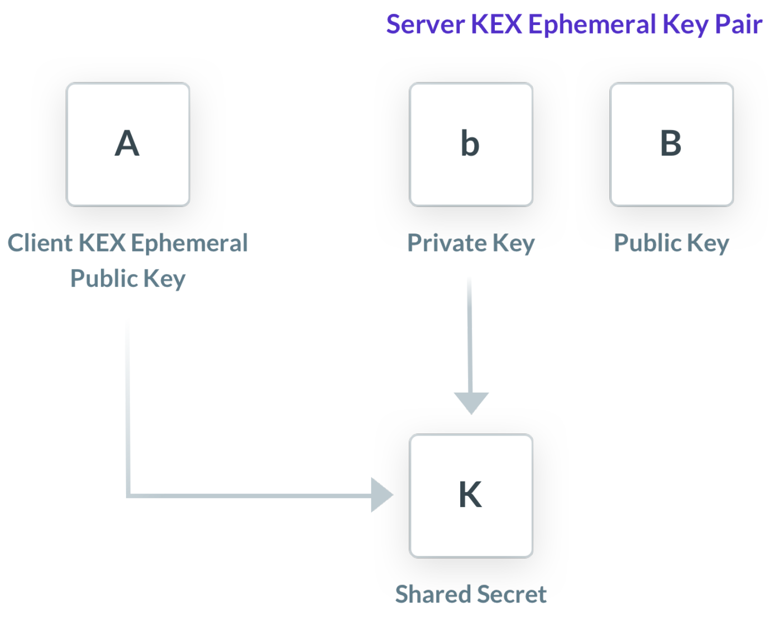

Server KEX Ephemeral Key Pair
A
b
B
Client KEX Ephemeral Public Key
Private Key
Public Key
K
Shared Secret

Figure 1: Shared secret creation

Authentication

During the key exchange (KEX), the client has authenticated the server, but the server has not yet authenticated the client. In most cases, asymmetric — or public-key — authentication is used by the client. This method involves two keys, a public and private key. Either can be used to encrypt a message, but the *other* must be used to decrypt:

- Data encrypted with the public key can only be decrypted with the private key
- Data encrypted with the private key can only be decrypted with the public key.

SSH public key authentication relies on asymmetric cryptographic algorithms that generate a pair of separate keys (a key pair), one "private" and the other "public". SSH Keys provide a level of authorization that can only be fulfilled by those who have ownership of the private key associated with the public key on the server.

If Bob encrypts a message with Alice's public key, only Alice's private key can decrypt the message. This principle is what allows the SSH protocol to authenticate identity.

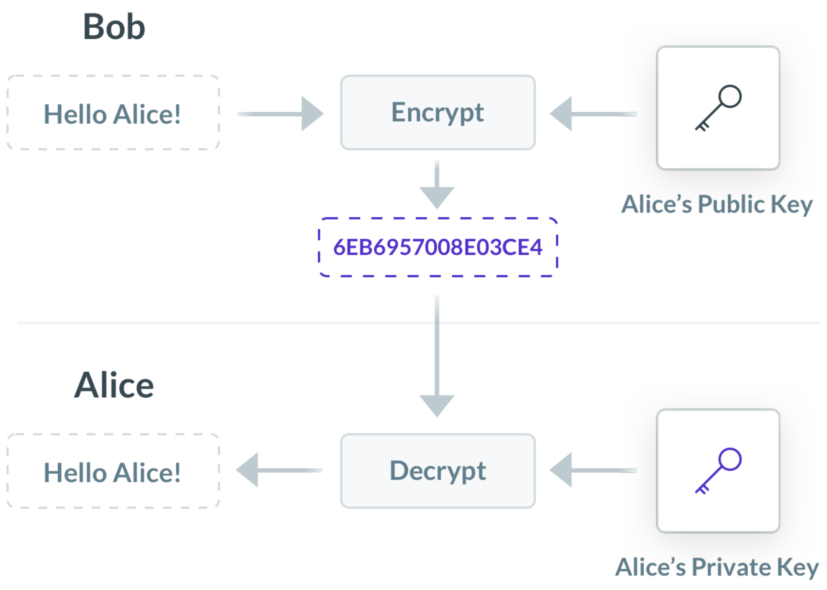


Figure 2: Only Alice's private key can decrypt a message signed with Alice's public key

If Alice (client) can decrypt Bob's (server) message, then it proves Alice is in possession of the paired private key. This is, in theory, how SSH keys authentication should work. Yet with the dynamic nature of infrastructure today, SSH keys are increasingly shared or managed improperly, compromising their integrity.

Asymmetric encryption algorithms

What makes asymmetric encryption powerful is that a private key can be used to derive a paired public key, but not the other way around. This principle is core to public-key authentication. If Alice had used a weak encryption algorithm that could be brute-forced by today's processing capabilities, a third party could derive Alice's private key using her public key. Protecting against a threat like this requires careful selection of the right encryption algorithm.

Three classes of algorithms are commonly used for asymmetric encryption:

1. RSA
2. DSA
3. Elliptic curve based algorithms.

Let's take a closer look at the mathematics behind each to determine its strength and integrity.

RSA: Integer factorization

First used in 1978, the RSA cryptography is based on the belief that factoring large semi-prime numbers is difficult by nature. Given that no general-purpose formula has been found to factor a compound number into its prime factors, there is a direct relationship between the size of the factors chosen and the time required to compute the solution. In other words, given a number ***n=p*q*** where ***p*** and ***q*** are sufficiently large prime numbers, it can be assumed that anyone who can factor ***n*** into its component parts is the only party that knows the values of ***p*** and ***q***. The same logic exists for public and private keys. In fact, ***p & q*** are necessary variables for the creation of a private key, and ***n*** is a variable for the subsequent public key.

DSA: Discrete logarithm problem & modular exponentiation

DSA follows a similar schema to RSA with public/private keypairs that are mathematically related. What makes DSA different from RSA is that DSA uses a different algorithm. It solves an entirely different problem using different elements, equations, and steps. The use of a randomly generated number, ***m***, is used with signing a message along with a private key, ***k***. This number ***m*** must be kept private.

ECDSA & EdDSA: Elliptic curve discrete logarithm problem

Algorithms using elliptic curves are also based on the assumption that there is no generally efficient solution to solving a discrete log problem. However, ECDSA/EdDSA and DSA differ in that DSA uses a mathematical operation known as modular exponentiation while ECDSA/EdDSA uses elliptic curves. The computational complexity of the discrete log problem allows both classes of algorithms to achieve the same level of security as RSA with significantly smaller keys.

Comparing encryption algorithms

Four main criteria guide the right algorithm choice:

- **Implementation** - Can you use a pre-existing library?
- **Compatibility** - Are there SSH clients that do not support a method?
- **Performance** - How long will it take to generate a sufficiently secure key?
- **Security** - Can the public key be derived from the private key?

RSA

Implementation	RSA libraries can be found for all major languages, including in-depth libraries (JS, Python, Go, Rust, C).
Compatibility	Usage of SHA-1 (OpenSSH) or public keys under 2048-bits may be unsupported.
Performance	Larger keys require more time to generate.
Security	Specialized algorithms like Quadratic Sieve and General Number Field Sieve exist to factor integers with specific qualities.

Time has been RSA's greatest ally and greatest enemy. First published in 1977, RSA has the widest support across all SSH clients and languages and has truly stood the test of time as a reliable key generation method. Subsequently, it has also been subject to Moore's Law for decades and key bit-length has grown in size. According to NIST standards, achieving 128-bit security requires a key with length 3072 bits whereas other algorithms use smaller keys. Bit security measures the number of trials required to brute-force a key. 128 bit security means 2^{128} trials to break.

DSA

Implementation	DSA was adopted by FIPS-184 in 1994. It has ample representation in major crypto libraries, similar to RSA.
Compatibility	While DSA enjoys support for PuTTY-based clients, OpenSSH 7.0 disables DSA by default.
Performance	Significant improvement in key generation times to achieve comparable security strengths, though recommended bit-length is the same as RSA.
Security	DSA requires the use of a randomly generated unpredictable and secret value that, if discovered, can reveal the private key.

In DSA, the use of a randomly generated number, *m*, is used with signing a message along with a private key, *k*. This number m must be kept privately. The value m is meant to be a nonce, which is a unique value included in many cryptographic protocols. However, the additional conditions of unpredictability and secrecy makes the nonce more akin to a key, and therefore extremely important. Not only is it difficult to ensure true randomness within a machine, but improper implementation can break encryption.

Choosing the encryption algorithm

Given the above, the choice is between RSA 2048⁄4096 and Ed25519, and the trade-off is between performance and compatibility:

- RSA is universally supported among SSH clients.
- EdDSA performs much faster and provides the same level of security with significantly smaller keys.

Ultimately, here's what really matters regarding that algorithm choice, in the words of Peter Ruppel: "The short answer to this is: as long as the key strength is good enough for the foreseeable future, it doesn't really matter. Because here we are considering a signature for authentication within an SSH session. The cryptographic strength of the signature just needs to withstand the current, state-of-the-art attacks."

ECDSA & EdDSA

ECDSA is an elliptic curve implementation of DSA. Functionally, where RSA and DSA require key lengths of 3072 bits to provide 128 bits of security, ECDSA can accomplish the same with only 256-bit keys. However, ECDSA relies on the same level of randomness as DSA, so the only gain is speed and length, not security.

In response to the desired speeds of elliptic curves and the undesired security risks, another class of curves has gained some notoriety. EdDSA solves the same discrete log problem as DSA/ECDSA, but uses a different family of elliptic curves known as the Edwards Curve (EdDSA uses a Twisted Edwards Curve). While offering slight advantages in speed over ECDSA, its popularity comes from an improvement in security. Rather than relying on a random number for the nonce value, EdDSA generates a nonce deterministically as a hash making it collision-resistant.

Taking a step back, the use of elliptic curves does not automatically guarantee some level of security. Not all curves are the same:

- Only a few curves have made it past rigorous testing.
- The PKI industry has slowly come to adopt Curve25519 in particular for EdDSA. Put together that makes the public-key signature algorithm, Ed25519.

Implementation	EdDSA is fairly new. Crypto++ and cryptlib do not currently support EdDSA.
Compatibility	Compatible with newer clients, Ed25519 has seen the largest adoption among the Edward Curves, though NIST also proposed Ed448 in their recent draft of SP 800-186.
Performance	Ed25519 is the fastest performing algorithm across all metrics. As with ECDSA, public keys are twice the length of the desired bit security.
Security	EdDSA provides the highest security level compared to key length. It also improves on the insecurities found in ECDSA.

Chapter 2 - Reducing your attack surface with an SSH proxy

To secure SSH access, it's good practice to use a proxy, i.e. a single access point which can forward clients to destination hosts. An SSH proxy is often called "jump server" or "jump host" or "bastion host" — which serves as the only gateway for access to your infrastructure, thereby reducing the size of any potential attack surface.

SSH jump server definition

An SSH jump server is a regular Linux server, accessible from the Internet, which is used as a gateway to access other Linux machines on a private network using the SSH protocol. Having a dedicated SSH access point also makes it easier to have an aggregated audit log of all SSH connections. The jump server name derives from the early days of SSH, when users had to SSH into a jump host and from there, had to type ssh again to "jump" to a destination host. Today, this can be done automatically using the ProxyJump option.

Two approaches to setting up an SSH jump server

This chapter will cover setting up an SSH jump server, through two distinct open source projects. Both are free, easy to install and configure, and are single-binary Linux daemons:

- A traditional SSH jump server using OpenSSH. The advantage of this method is that your servers already have OpenSSH pre-installed.
- A modern approach using Teleport, a newer open source alternative to OpenSSH. Teleport's advantage over OpenSSH is that in addition to SSH it supports other access protocols (Kubernetes, popular databases, etc) and natively integrates with identity-based access.

Having a dedicated SSH jump server (i.e., one that doesn't host any other publicly accessible software on it) is a good security practice. In contrast, it is bad practice to allow users to log into a jump server directly. There are a few reasons why:

- Inadvertently updating the jump server configuration.
- Using the jump server machine for other tasks.
- Making copies of keys used to access destination servers.

When using OpenSSH It is also a good idea to change the default TCP port on the SSH jump server from 22 to something else, as this can reduce the number of brute force attacks made against the default port. Teleport does this by default.

Assumptions made in the two examples below

The following walkthroughs for configuring an SSH jump server using the two above-mentioned open-source projects make the following naming assumptions:

- The example organization domain is example.com
- The DNS name of the jump server is going to be proxy.example.com

It's also assumed that proxy.example.com is the only machine accessible from the Internet.

OpenSSH

This SSH server comes bundled by default with most Linux distributions, and there's nearly a 100% chance you already have it installed. If the server is accessible via proxy.example.com then you can access other servers behind the same NAT boundary via -J command line flag, i.e. on the client:

```
$ ssh -J proxy.example.com 10.2.2.1
```

In the example above, the jump server is used to access another host on an AWS VPC with an address of `10.2.2.1.` So far, this looks pretty easy.

To avoid typing `-J proxy.example.com` all the time, you can update your client's SSH configuration in `~/.ssh/config` with the following:

```
Host 10.2.2.*
    ProxyJump proxy.example.com
```

Now, when a user types `ssh 10.2.2.1` their SSH client will not even try to resolve `10.2.2.1` locally, but instead will establish a connection to `proxy.example.com` which will forward it to `10.2.2.1` within its VPC.

Next, we need to harden the server configuration a bit by disabling interactive SSH sessions on the jump server for regular users, but leaving it turned on for the administrators. To do this, update the `sshd` configuration, usually in `/etc/ssh/sshd_config` with the following:

```
# Do not let SSH clients do anything except be forwarded to the destination:
PermitTTY no
X11Forwarding no
PermitTunnel no
GatewayPorts no
ForceCommand /sbin/nologin
```

The example above will work for Debian and its derivatives, we advise to verify the existence of `/sbin/nologin`.

This will work for as long as the jump server has accounts for all SSH users, which is inconvenient. Instead, consider creating a separate user account on the jump server dedicated to "jumping users". Let's call it `jumpuser` and update the configuration:

```
Match User jumpuser
  PermitTTY no
  X11Forwarding no
  PermitTunnel no
  GatewayPorts no
  ForceCommand /usr/sbin/nologin
```

And the users will have to update their client SSH configuration with:

```
Host 10.2.2.*
    ProxyJump jumpuser@proxy.example.com
```

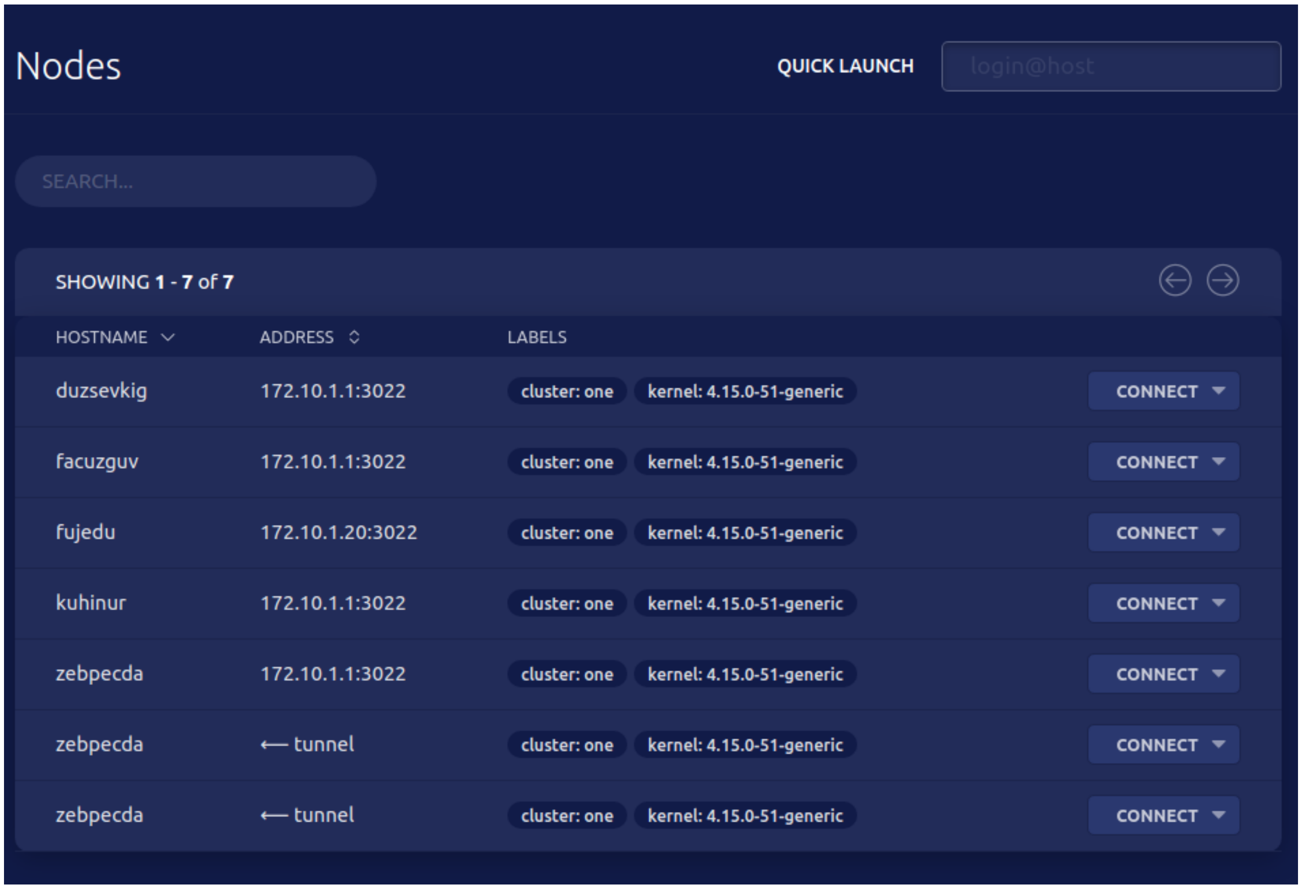
For more information on how to fine-tune SSH jump configuration to your particular situation, `consult man ssh_config and man sshd_config.`

Needless to say, the setup above works only when the public SSH keys are properly distributed not only between clients and the jump server, but also between the clients and the destination servers.

Teleport

Teleport is a much newer SSH server, which was released in 2016. Unlike OpenSSH, Teleport is a highly opinionated SSH implementation:

- **Teleport insists on using an SSH proxy by default**, and its SSH proxy has a web-based interface, allowing users to SSH using a browser.
- Teleport, unlike traditional SSH servers, **eliminates the need to maintain "inventories" of servers**, as it offers a live introspection, i.e. you can list all online servers behind a proxy as shown in this screenshot:



SHOWING 1 - 7 of 7

HOSTNAME	ADDRESS	LABELS	
duzsevkig	172.10.1.1:3022	cluster: one kernel: 4.15.0-51-generic	CONNECT
facuzguv	172.10.1.1:3022	cluster: one kernel: 4.15.0-51-generic	CONNECT
fujedu	172.10.1.20:3022	cluster: one kernel: 4.15.0-51-generic	CONNECT
kuhinur	172.10.1.1:3022	cluster: one kernel: 4.15.0-51-generic	CONNECT
zebpecda	172.10.1.1:3022	cluster: one kernel: 4.15.0-51-generic	CONNECT
zebpecda	⟵ tunnel	cluster: one kernel: 4.15.0-51-generic	CONNECT
zebpecda	⟵ tunnel	cluster: one kernel: 4.15.0-51-generic	CONNECT

Figure 3: Teleport screenshot showing real-time view of all online servers behind a proxy

In addition to having a modern proxy functionality, Teleport offers a few advantages over traditional SSH:

- **Does not use SSH keys** and instead defaults to identity-based access via SSH certificates. This removes the need for key management and makes SSH servers completely stateless and configuration-free.
- **Supports other protocols in addition to SSH**, so the same "jump host" can be used to access other resources behind NAT, such as Windows servers, popular databases, Kubernetes clusters or even internal applications via HTTP(s).
- **Does not rely on Linux users for authentication**; instead, Teleport maintains a separate database of users or can integrate with a single sign-on with other identity providers such as GitHub, Google Apps, or enterprise options such as Okta and Active Directory.

- **Supports role-based access control (RBAC)**. Teleport extends SSH by allowing users to have roles.
- **Centralized audit log**. Teleport natively supports a single unified audit log for the entire server fleet across all environments.
- **Edge deployments**. Teleport allows connectivity into servers that are running in untrusted or public networks. This works because a Teleport SSH server can maintain a permanent reverse tunnel to a proxy, allowing users to SSH into IoT devices "in the wild".

Teleport always comes with a proxy (i.e. the same thing as a "jump server") and there are no special instructions for setting it up.

Choosing OpenSSH or Teleport

Below are tips on choosing among these two open source projects to set up an SSH jump server:

Use OpenSSH if:

- The number of servers or/and users in your organization is small
- You need a jump host setup quickly and do not have much time to learn new tech

Use Teleport if:

- Your server fleet or the size of your team is growing
- You need to connect to servers located "in the wild", i.e. not restricted to a single VPC
- You want to unify access across multiple protocols using a single tool.

Chapter 3 - Using industry best practices

This chapter, written with OpenSSH users in mind, discusses three industry best practices that can improve your SSH model' security without requiring you to deploy a new application or make any major changes to user experience. These best practices, covered in more detail below, are:

1. Use SSH certificates
2. Enforce the use of bastion hosts
3. Add 2-factor authentication to your SSH logins

SSH certificates

Most people can agree that using public key authentication for SSH is generally better than using passwords. Nobody ever types in a private key, so it can't be keylogged or observed over your shoulder. SSH keys have their own issues, however.

While valid in theory, key-based authentication is probably not the best approach to SSH security in practice. Indeed, if not properly "managed", SSH keys can be no safer than passwords. SSH key management can get complicated because SSH key-based authentication simply does not scale. Rather than seeking a "SSH key management" solution, it's best practice to use short-lived, automatically expiring SSH certificates.

SSH certificates are the next level up from SSH keys. OpenSSH has supported their use since OpenSSH 5.4 which was released back in 2010. An SSH certificate is simply a public key signed by a well-known, trusted entity called a certificate authority ("CA").

With SSH certificates, you generate a certificate authority (CA) and then use this to issue and cryptographically sign certificates which can authenticate users to hosts, or hosts to users. A certificate authority is the ultimate grantor of trust in an organization. This means that copying keys around is no longer necessary; users and servers simply must agree on which CA to trust. You can generate a keypair using the ssh-keygen command, like this:

```
$ ssh-keygen -t rsa -b 4096 -f host_ca -C host_ca
Generating public/private rsa key pair.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in host_ca.
Your public key has been saved in host_ca.pub.
The key fingerprint is:
SHA256:tltbnMalWg+skhm+VlGLd2xHiVPozyuOPl34WypdEO0 host_ca
The key's randomart image is:
+---[RSA 4096]----+
|              +o.|
|            .+..o|
|           o.o.+ |
|          o o.= E|
|        S  o o=o |
|       ....+ = +.|
|       ..=. %.o.o|
|        *o Oo=.+.|
|       .oo=ooo+..|
+----[SHA256]-----+

$ ls -l
total 8
-rw-------. 1 gus gus 3381 Mar 19 14:30 host_ca
-rw-r--r--. 1 gus gus  737 Mar 19 14:30 host_ca.pub
```

The `host_ca` file is the host CA's private key and should be protected. Don't give it out to anyone, don't copy it anywhere, and make sure that as few people have access to it as possible. Ideally, it should live on a machine which doesn't allow direct access and all certificates should be issued by an automated process.

In addition, it's best practice to generate and use two separate CAs - one for signing host certificates, one for signing user certificates. This is because you don't want the same processes that add hosts to your fleet to also be able to add users (and vice versa). Using separate CAs also means that in the event of a private key being compromised, you only need to reissue the certificates for either your hosts or your users, not both at once.

As such, we'll also generate a `user_ca` with this command:

```
$ ssh-keygen -t rsa -b 4096 -f user_ca -C user_ca
```

The `user_ca` file is the user CA's private key and should also be protected in the same way as the host CA's private key.

Issuing host certificates (to authenticate hosts to users)

In this example, we'll generate a new host key (with no passphrase), then sign it with our CA. You can also sign the existing SSH host public key if you'd prefer not to regenerate a new key by copying the file (`/etc/ssh/ssh_host_rsa_key.pub`) from the server, signing it on your CA machine, and then copying it back.

```
$ ssh-keygen -f ssh_host_rsa_key -N '' -b 4096 -t rsa

$ ls -l
-rw------- 1 ec2-user ec2-user 3247 Mar 17 14:49 ssh_host_rsa_key
-rw-r--r-- 1 ec2-user ec2-user  764 Mar 17 14:49 ssh_host_rsa_key.pub

$ ssh-keygen -s host_ca -I host.example.com -h -n host.example.com -V +52w
ssh_host_rsa_key.pub
Enter passphrase: # the passphrase used for the host CA
Signed host key ssh_host_rsa_key-cert.pub: id "host.example.com" serial 0
for host.example.com valid from 2020-03-16T15:00:00 to 2021-03-15T15:01:37

$ ls -l
-rw------- 1 ec2-user ec2-user 3247 Mar 17 14:49 ssh_host_rsa_key
-rw-r--r-- 1 ec2-user ec2-user 2369 Mar 17 14:50 ssh_host_rsa_key-cert.pub
-rw-r--r-- 1 ec2-user ec2-user  764 Mar 17 14:49 ssh_host_rsa_key.pub
ssh_host_rsa_key-cert.pub
```
contains the signed host certificate.

Here's an explanation of the flags used:

- `-s host_ca`: specifies the filename of the CA private key that should be used for signing.
- `-I host.example.com`: the certificate's identity - an alphanumeric string that will identify the server. I recommend using the server's hostname. This value can also be used to revoke a certificate in future if needed.
- `-h: specifies` that this certificate will be a host certificate rather than a user certificate.
- `-n host.example.com`: specifies a comma-separated list of principals that the certificate will be valid for authenticating - for host certificates, this is the hostname used to connect to the server. If you have DNS set up, you should use the server's FQDN (for example host.example.com) here. If not, use the hostname that you will be using in an `~/.ssh/config` file to connect to the server.
- `-V +52w: specifies` the validity period of the certificate, in this case 52 weeks (one year). Certificates are valid forever by default - expiry periods for host certificates are highly recommended to encourage the adoption of a process for rotating and replacing certificates when needed.

To see the options that a given certificate was signed with, use `ssh-keygen -L`:

```
$ ssh-keygen -L -f user-key-cert.pub
user-key-cert.pub:
        Type: ssh-rsa-cert-v01@openssh.com user certificate
        Public key: RSA-CERT
SHA256:egWNu5cUZaqwm76zoyTtktac2jxKktj30Oi/ydrOqZ8
        Signing CA: RSA SHA256:tltbnMalWg+skhm+VlGLd2xHiVPozyuOPl34WypdEO0
(using ssh-rsa)
        Key ID: "gus@goteleport.com"
        Serial: 0
        Valid: from 2020-03-19T16:33:00 to 2020-03-20T16:34:54
        Principals:
                ec2-user
                gus
        Critical Options: (none)
        Extensions:
                permit-X11-forwarding
```

```
                 permit-agent-forwarding
                 permit-port-forwarding
                 permit-pty
                Permit-user-rc
```

Configuring SSH to use host certificates

You also need to tell the server to use this new host certificate. Copy the three files you just generated to the server, store them under the `/etc/ssh` directory, set the permissions to match the other files there, then add this line to your `/etc/ssh/sshd_config` file:

```
HostCertificate /etc/ssh/ssh_host_rsa_key-cert.pub
```

Once this is done, restart sshd with `systemctl restart sshd`.

Your server is now configured to present a certificate to anyone who connects. For your local ssh client to make use of this (and automatically trust the host based on the certificate's identity), you will also need to add the CA's public key to your known_hosts file.

You can do this by taking the contents of the `host_ca.pub` file, adding `@cert-authority *.example.com` to the beginning, then appending the contents to `~/.ssh/known_hosts`:

```
@cert-authority *.example.com ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAACAQDwiOso0Q4W+KKQ4OrZZ1o1X7g3yWcmAJtySILZSwo1GXBKgurV4jm
mBN5RsHetl98QiJq64e8oKX1vGR251afalWu0w/iW9jL0isZrPrmDg/p6Cb6yKnreFEaDFocDhoiIcbUiIm
IWcp9PJXFOK1Lu8afdeKWJA2f6cC4lnAEq4sA/Phg4xfKMQZUFG5sQ/Gj1StjIXi2RYCQBHFDzzNm0Q5uB4
hUsAYNqbnaiTI/pRtuknsgl97xK9P+rQiNfBfPQhsGeyJzT6Tup/KKlxarjkMOlFX2MUMaAj/cDrBSzvSrf
OwzkqyzYGHzQhST/lWQZr4OddRszGPO4W5bRQzddUG8iC7M6U4llUxrb/H5QOkVyvnx4Dw76MA97tiZItSG
zRPblU4S6HMmCVpZTwva4LLmMEEIk1lW5HcbB6AWAc0dFE0KBuusgJp9MlFkt7mZkSqnim8wdQApal+E3p1
3d0QZSH3b6eB3cbBcbpNmYqnmBFrNSKkEpQ8OwBnFvjjdYB7AXqQqrcqHUqfwkX8B27chDn2dwyWb3AdPMg
1+j3wtVrwVqO9caeeQ1310CNHIFhIRTqnp2ECFGCCy+EDSFNZM+JStQoNO5rMOvZmecbp35XH/UJ5IHOkh9
wE5TBYIeFRUYoc2jHNAuP2FM4LbEagGtP8L5gSCTXNRM1EX2gQ== host_ca
```

The value `*.example.com` is a pattern match, indicating that this certificate should be trusted for identifying any host which you connect to that has a domain of `*.example.com` - such as `host.example.com` above. This is a comma-separated list of applicable hostnames for the certificate, so if you're using IP addresses or SSH config entries here, you can change this to something like `host1,host2,host3 or 1.2.3.4,1.2.3.5` as appropriate.

Once this is configured, remove any old host key entries for `host.example.com` in your `~/.ssh/known_hosts` file, and start an ssh connection. You should be connected straight to the host without needing to trust the host key. You can check that the certificate is being presented correctly with a command like this:

```
$ ssh -vv host.example.com 2>&1 | grep "Server host certificate"
debug1: Server host certificate: ssh-rsa-cert-v01@openssh.com
SHA256:dWi6L8k3Jvf7NAtyzd9LmFuEkygWR69tZC1NaZJ3iF4, serial 0 ID "host.example.com"
CA ssh-rsa SHA256:8gVhYAAW9r2BWBwh7uXsx2yHSCjY5OPo/X3erqQi6jg valid from
2020-03-17T11:49:00 to 2021-03-16T11:50:21
debug2: Server host certificate hostname: host.example.com
```

At this point, you could continue by issuing host certificates for all hosts in your estate using your host CA. The benefit of doing this is twofold: you no longer need to rely on the insecure trust on first use (TOFU) model for new hosts, and if you ever redeploy a server and therefore change the host key for a certain hostname, your new host could automatically present a signed host certificate and avoid the dreaded **`WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!`** message.

Issuing user certificates (to authenticate users to hosts)

In this example, we'll generate a new user key and sign it with our user CA. It's up to you whether you use a passphrase or not.

```
$ ssh-keygen -f user-key -b 4096 -t rsa

$ ls -l
-rw-r--r--. 1 gus gus  737 Mar 19 16:33 user-key.pub
-rw-------. 1 gus gus 3369 Mar 19 16:33 user-key

$ ssh-keygen -s user_ca -I gus@goteleport.com -n ec2-user,gus -V +1d user-key.pub
Enter passphrase: # the passphrase used for the user CA
```

```
Signed user key user-key-cert.pub: id "gus@goteleport.com" serial 0 for
ec2-user,gus valid from 2020-03-19T16:33:00 to 2020-03-20T16:34:54

$ ls -l
-rw-------. 1 gus gus 3369 Mar 19 16:33 user-key
-rw-r--r--. 1 gus gus 2534 Mar 19 16:34 user-key-cert.pub
-rw-r--r--. 1 gus gus  737 Mar 19 16:33 user-key.pub
```

`user-key-cert.pub` contains the signed user certificate. You'll need both this and the private key (user-key) for logging in.

Here's an explanation of the flags used:

- `-s user_ca`: specifies the CA private key that should be used for signing
- `-I gus@goteleport.com`: the certificate's identity, an alphanumeric string that will be visible in SSH logs when the user certificate is presented. I recommend using the email address or internal username of the user that the certificate is for - something which will allow you to uniquely identify a user. This value can also be used to revoke a certificate in future if needed.
- `-n ec2-user,gus`: specifies a comma-separated list of principals that the certificate will be valid for authenticating, i.e. the *nix users which this certificate should be allowed to log in as. In our example, we're giving this certificate access to both `ec2-user` and `gus`.
- `-V +1d`: specifies the validity period of the certificate, in this case +1d means 1 day. Certificates are valid forever by default, so using an expiry period is a good way to limit access appropriately and ensure that certificates can't be used for access perpetually.

If you need to see the options that a given certificate was signed with, you can use `ssh-keygen -L`:

```
$ ssh-keygen -L -f user-key-cert.pub
user-key-cert.pub:
        Type: ssh-rsa-cert-v01@openssh.com user certificate
        Public key: RSA-CERT SHA256:egWNu5cUZaqwm76zoyTtktac2jxKktj30Oi/ydrOqZ8
        Signing CA: RSA SHA256:tltbnMalWg+skhm+VlGLd2xHiVPozyuOPl34WypdEO0 (using
ssh-rsa)
        Key ID: "gus@goteleport.com"
        Serial: 0
        Valid: from 2020-03-19T16:33:00 to 2020-03-20T16:34:54
        Principals:
                ec2-user
                gus
        Critical Options: (none)
        Extensions:
                permit-X11-forwarding
                permit-agent-forwarding
                permit-port-forwarding
                permit-pty
                permit-user-rc
```

Configuring SSH for user certificate authentication

Once you've signed a certificate, you also need to tell the server that it should trust certificates signed by the user CA. To do this, copy the `user_ca.pub` file to the server and store it under `/etc/ssh,` fix the permissions to match the other public key files in the directory, then add this line to `/etc/ssh/sshd_config`:

```
TrustedUserCAKeys /etc/ssh/user_ca.pub
```

Once this is done, restart `sshd` with systemctl restart `sshd`.

Your server is now configured to trust anyone who presents a certificate issued by your user CA when they connect. If you have a certificate in the same directory as your private key (specified with the `-i flag`, for example `ssh -i /home/gus/user-key ec2-user@host.example.com`), it will automatically be used when connecting to servers.

Checking logs

If you look in your server's sshd log (for example, by running journalctl -u sshd), you will see the name of the certificate being used for authentication, along with the fingerprint of the signing CA:

```
sshd[14543]: Accepted publickey for ec2-user from 1.2.3.4 port 53734 ssh2: RSA-CERT
ID gus@goteleport.com (serial 0) CA RSA
SHA256:tltbnMalWg+skhm+VlGLd2xHiVPozyuOPl34WypdEO0
```

If the user tries to log in as a principal (user) which they do not have permission to use (for example, their certificate grants `ec2-user` but they try to use root), you'll see this error in the logs:

```
sshd[14612]: error: key_cert_check_authority: invalid certificate
sshd[14612]: error: Certificate invalid: name is not a listed principal
```

If the certificate being presented has expired, you'll see this error in the logs:

```
sshd[14240]: error: key_cert_check_authority: invalid certificate
sshd[14240]: error: Certificate invalid: expired
```

One way that you could make further use of user certificates is to set up a script which will use your CA to issue a certificate to log into production servers, valid only for the next two hours. Every use of this script or process could add logs as to who requested a certificate and embed their email address into the certificate. After this is done, every time the user uses that certificate to access a server (regardless of which principal they log in as), their email address will be logged. This can provide a useful part of an audit trail.

There are many other useful things you can do with SSH certificates, such as forcing a specific command to be run when presenting a certain certificate, or denying the ability to forward ports, X11 traffic or SSH agents. Take a look at man ssh-keygen for some ideas.

Enforce the use of a bastion host

Another way to improve your SSH security is to enforce the use of a bastion host - a server which is specifically designed to be the only gateway for access to your infrastructure. Lessening the size of any potential attack surface through the use of firewalls enables you to keep a better eye on who is accessing what.

Switching to the use of a bastion host doesn't have to be an arduous task, especially if you're using SSH certificates. By setting up your local `~/.ssh/config` file, you can automatically configure all connections to hosts within a certain domain to go through the bastion.

Here's a very quick example of how to force SSH access to any host in the example.com domain to be routed through a bastion host, `bastion.example.com`:

```
Host *.example.com
    ProxyJump bastion.example.com
    IdentityFile ~/user-key

Host bastion.example.com
    ProxyJump none
    IdentityFile ~/user-key
```

To make this even simpler, if you add `user-key` to your local `ssh-agent` with `ssh-add user-key`, you can also remove the `IdentityFile` entries from the SSH config file.

Once you're using the bastion host for your connections, you can use `iptables` (or another *nix firewall configuration tool of your choosing) on servers behind the bastion to block all other incoming SSH connections. Here's a rough example using `iptables`:

```
$ iptables -A INPUT -m state --state RELATED,ESTABLISHED -j ACCEPT
$ iptables -A INPUT -p tcp --dport 22 -s <public IP of the bastion> -j
ACCEPT
$ iptables -A INPUT -p tcp --dport 22 -j DROP
```

It's a good idea to leave a second SSH session connected to the bastion while running these commands so that if you inadvertently input the wrong IP address or command, you should still have working access to the bastion to fix it via the already-established connection.

Add 2-factor authentication to your SSH logins

Multi-factor authentication makes it more difficult for bad actors to log into your systems by enforcing the need for two different "factors" or methods to be able to successfully authenticate.

This usually comes down to needing both "something you know" (like a password, or SSH certificate in our example) and "something you have" (like a token from an app installed on your phone, or an SMS with a unique code). One other possibility is requiring the use of "something you are" - for example a fingerprint, or your voice.

Teleport comes with multi-factor authentication built-in and no special configuration is required.

In this chapter we'll focus on configuring OpenSSH. We'll install the google-authenticator pluggable authentication module, which will require users to input a code from the Google Authenticator app on their phone in order to log in successfully. You can download the app for iOS here and Android here.

As a general note, it's always important to consider the user experience when enforcing security measures. If your measures are too draconian then users may attempt to find ways to defeat and work around them, which will eventually reduce the overall security of your systems and lead to the creation of back doors. To give our users a reasonable experience in this example, we are only going to require 2-factor authentication to be able to log into our bastion host.

Once authenticated there, users should be able to log into other hosts simply by using their valid SSH certificate. This combination should give an acceptable level of security without interfering too much with user workflows. With this in mind, however, it is always prudent and appropriate to enforce extra security measures in specific environments which contain critical production data or sensitive information.

Install google-authenticator

On RHEL/CentOS based systems, you can install the google-authenticator module from the EPEL repository:

```
$ sudo yum install
https://dl.fedoraproject.org/pub/epel/epel-release-latest-7.noarch.rpm # for
RHEL/CentOS 7, change for other versions
$ sudo yum install google-authenticator
```

For Debian`/Ubuntu-`based systems, this is available as the `libpam-google-authenticator` package:

```
$ sudo apt-get install libpam-google-authenticator
```

The `google-authenticator` module has many options you can set which are documented here. In the interest of saving time, we are going to use some sane defaults in this example: disallow reuse of the same token twice, issue time-based rather than counter-based codes, and limit the user to a maximum of three logins every 30 seconds. To set up Google 2-factor authentication with these settings, a user should run this command:

```
$ google-authenticator -d -f -t -r 3 -R 30 -W
```

You can also run `google-authenticator` with no flags and answer some prompts to set up interactively if you prefer.

This will output a QR code that the user can scan with the app on their phone, plus some backup codes which they can use if they lose access to the app. These codes should be stored offline in a secure location.

Scan the generated QR code for your user now with the Google Authenticator app and make sure that you have a 6-digit code displayed. If you need to edit or change any settings in future, or remove the functionality completely, the configuration will be stored under `~/.google_authenticator`.

Configure PAM for 2-factor authentication

To make the system enforce the use of these OTP (one-time password) codes, we'll first need to edit the PAM configuration for the sshd service (`/etc/pam.d/sshd`) and add this line to the end of the file:

```
auth required pam_google_authenticator.so nullok
```

The `nullok` at the end of this line means that users who don't have a second factor configured yet will still be allowed to log in so that they can set one up. Once you have 2-factor authentication set up for all your users, you should remove `nullok` from this line to properly enforce the use of a second factor.

We also need to change the default authentication methods so that SSH won't prompt users for a password if they don't present a 2-factor token. These changes are also made in the

`/etc/pam.d/sshd` file:

- On RHEL/CentOS-based systems, comment out `auth substack password-auth` by adding a # to the beginning of the line: `#auth substack password-auth`
- On Debian`/Ubuntu`-based systems, comment out `@include common-auth` by adding a # to the beginning of the line: `#@include common-auth`

Save the `/etc/pam.d/sshd` file once you're done.

Configure SSH for 2-factor authentication

We also need to tell SSH to require the use of 2-factor authentication. To do this, we make a couple of changes to the `/etc/ssh/sshd_config` file.

Firstly, we need to change `ChallengeResponseAuthentication no` to `ChallengeResponseAuthentication yes` to allow the use of PAM for credentials.

We also need to set the list of acceptable methods for authentication by adding this line to the end of the file (or editing the line if it already exists):

```
AuthenticationMethods publickey,keyboard-interactive
```

This tells SSH that it should require both a public key (which we are going to be satisfying using an SSH certificate) and a keyboard-interactive authentication (which will be provided and satisfied by the sshd PAM stack). Save the `/etc/ssh/sshd_config` file once you're done.

At this point, you should restart sshd with `systemctl` restart `sshd`. Make sure to leave an SSH connection open so that you can fix any errors if you need to. Restarting SSH will leave existing connections active, but new connections may not be allowed if there is a configuration problem.

Test it out

Connect to your bastion host directly and you should see a prompt asking you for your 2-factor code:

```
$ ssh bastion.example.com
Verification code:
```

Type the code presented by your Google Authenticator app and your login should proceed normally.

If you check the sshd log with `journalctl -u sshd`, you should see a line indicating that your login succeeded:

```
Mar 23 16:51:13 ip-172-31-33-142.ec2.internal sshd[29340]: Accepted
keyboard-interactive/pam for gus from 1.2.3.4 port 42622 ssh2
```

Conclusion

This tech paper covered three practical approaches to securing SSH access:

- Comparing SSH key algorithms (RSA, DSA, ECDSA, or EdDSA) to select the right SSH key;
- Setting up a jump server to reduce your attack surface with through openSSH or through Teleport
- Adopting industry best practices to improve the security of your SSH model, such as using SSH certificates, enforcing the use of bastion hosts and adding 2-factor authentication to your SSH logins

Though SSH is a widely adopted and powerful security tool that protects access to mission-critical systems, it can become a security liability if improperly managed.

As a newer SSH server built-in with many industry best practices, Teleport (available in both open source and paid versions) provides a viable alternative. It offers more visibility by allowing a live view into all online servers behind a proxy, uses secure and flexible SSH certificates rather than SSH keys, supports other protocols in addition to SSH; and can integrate with a single sign-on with other identity providers.