Keybase SSH CA Bot Documentation

keybase.io

Jun 20, 2020

Contents

CHAPTER 1

Introduction

This repository provides the tooling to control SSH access to servers (without needing to install anything on them) based simply on cryptographically provable membership in Keybase teams.

SSH supports a concept of certificate authorities (CAs) where you can place a single public key on the server, and the SSH server will allow any connections with keys signed by the CA cert. This is how a lot of large companies manage SSH access securely; users can be granted SSH access to servers without having to change the keys that are deployed on the server.

This repo provides the pieces for anyone to build this workflow on top of Keybase:

1. generation scripts and a guide to set up the Keybase team and server ssh configuration
2. a wrapper around ssh (`kssh`) for any end user to get authenticated using the certificate authority
3. a chatbot (`keybaseca`) which listens in a Keybase team for `kssh` requests. If the requester is in the team, the bot will sign the request with an expiring signature (e.g. 1 hour), and then the provisioned server should authenticate as usual.

Removing a user's ability to access a server is as simple as removing them from the Keybase team.

CHAPTER 2

Getting Started

kssh allows you to define realms of servers where access is granted based off of membership in different teams. Imagine that you have a staging environment that everyone should be granted access to and a production environment that you want to restrict access to a smaller group of people. For this exercise we'll also set up a third realm that grants root access to all machines.

Start by creating a new Keybase user to use for the CA chatbot:

```
keybase signup      # Creates a new Keybase user to use for the SSH CA bot
keybase paperkey    # Generate a new paper key
```

Note that this system will not work if you attempt to use the same user for the CA chatbot as for kssh. It is required to use distinct users.

Then create `{TEAM}.ssh.staging`, `{TEAM}.ssh.production`, `{TEAM}.ssh.root_everywhere` as new Keybase subteams and add the bot to those subteams. Add users to those subteams based off of the permissions you wish to grant different users

On a secured server (note that this server only needs docker installed) that you wish to use to run the CA chatbot:

```
git clone git@github.com:keybase/bot-sshca.git
cd bot-sshca/docker/
cp env.list.example env.list
nano env.list       # Fill in the values including the previously generated paper key
make generate       # Generate a new CA key
```

Running `make generate` will output a list of configuration steps to run on each server you wish to use with the CA chatbot. These commands create a new user to use with kssh (the `developer` user), add the CA's public key to the server, and configure the server to trust the public key.

Now you must define a mapping between Keybase teams and the users on the servers that members of those teams are allowed to access. If you wish to make the user `foo` on your server available to anyone in `team.ssh.bar`, create the file `/etc/ssh/auth_principals/foo` with contents `team.ssh.bar`.

More concretely following the current example setup:

- For each server in your staging environment:

1. Create the file `/etc/ssh/auth_principals/root` with contents `{TEAM}.ssh.root_everywhere`
2. Create the file `/etc/ssh/auth_principals/developer` with contents `{TEAM}.ssh.staging`

- For each server in your production environment:
  1. Create the file `/etc/ssh/auth_principals/root` with contents `{TEAM}.ssh.root_everywhere`
  2. Create the file `/etc/ssh/auth_principals/developer` with contents `{TEAM}.ssh.production`

Now on the server where you wish to run the chatbot, start the chatbot itself:

```
make serve    # Runs inside of docker for ease of use
```

You can confirm that the bot is running by sending the message `ping @bot_username` in any of the configured team chat channels (if `CHAT_CHANNEL` is configured, the message must be sent in that specific channel). The bot should reply with `pong @your_username`.

Now you can download the kssh binary and start SSHing! See https://github.com/keybase/bot-sshca/releases to download the most recent version of kssh for your platform.

```
sudo mv kssh-{platform} /usr/local/bin/kssh
sudo chmod +x /usr/local/bin/kssh

kssh developer@staging-server-ip        # If in {TEAM}.ssh.staging
kssh developer@production-server-ip     # If in {TEAM}.ssh.production
kssh root@server                        # If in {TEAM}.ssh.root_everywhere
```

We recommend building kssh yourself and distributing the binary among your team (perhaps in Keybase Files!).

2.1 Updating environment variables

If you update any environment variables, it is necessary to restart the keybaseca service. This can be done by running `make restart`. Note that it is not required to re-run `make generate`.

Note that this means `kssh` will not work for a brief period of time while the container restarts.

CHAPTER 3

Advanced Configuration

The SSH CA bot is configured via environment variables. This documents lists the different environment variables used by the bot and their purpose.

3.1 Environment Variables

3.1.1 TEAMS

The `TEAMS` environment variable configures which teams the SSH CA bot will use to grant SSH access.

Examples:

```
export TEAMS="team.ssh"
export TEAMS="team.ssh.prod"
export TEAMS="team.ssh.prod,team.ssh.staging,team.ssh.root_everywhere"
```

3.1.2 CA_KEY_LOCATION

The `CA_KEY_LOCATION` environment variable configures where the CA bot will store the CA key. It is recommended to ensure that the CA key is stored in a secure location. Defaults to `/mnt/keybase-ca-key`.

Examples:

```
export CA_KEY_LOCATION="/etc/cakey"
export CA_KEY_LOCATION="~/secure/cakey"
```

3.1.3 KEY_EXPIRATION

The `KEY_EXPIRATION` environment variable configures the validity length of keys signed by the bot. A key provisioned via kssh is valid for this length of time before kssh will automatically reprovision another key. It is recom-

mended to keep the key expiration window to a relatively short period of time. By default, signed key s expire after one hour. Valid formats are +30m, +1h, +5h, +1d, +3d, +1w, etc

Examples:

```
export KEY_EXPIRATION="+2h"
export KEY_EXPIRATION="+10m"
export KEY_EXPIRATION="+1w"     # not recommended to set it to a time period this long
```

3.1.4 LOG_LOCATION

The `LOG_LOCATION` environment variable configures where logs from the CA bot will be stored. It is recommended to store these logs in a secure location for audit purposes. One potential option is storing logs in a KBFS subteam dedicated to admins. If not set, logs go to stdout.

Examples:

```
export LOG_LOCATION="/keybase/team/teamname.ssh.admin/keybaseca_audit.log"
```

3.1.5 STRICT_LOGGING

The `STRICT_LOGGING` environment variable defines the behavior of the bot if it fails to save an audit log entry. By default, if the CA bot fails to write a log to a file it will simply send it to stdout. If it is critical to maintain correct audit logs, the `STRICT_LOGGING` option will cause the CA bot to panic and shutdown if it is unable to save logs.

Examples:

```
export STRICT_LOGGING="true"
export STRICT_LOGGING="false"
```

3.1.6 CHAT_CHANNEL

The `CHAT_CHANNEL` environment variable controls where communication between the bot and users will take place. It specifies a specific team and channel. This may be useful if subteams are being used for more purposes than just SSH access. For example, one could use team.prod to grant SSH production access and for a secret sharing bot used to share other credentials. One would then want to configure the CA bot to use a separate channel (eg #ssh-provision) for provision requests so that the general channel could be used for lower volume purposes. Note that this means that all users of the SSH bot must have access to this channel.

Examples:

```
export CHAT_CHANNEL="team.prod#ssh-provision"
export CHAT_CHANNEL="team.ssh_bot#general"
```

3.1.7 Announcement

The `ANNOUNCEMENT` environment variable contains a string that will be announced in all of the configured teams when the bot is started. This is useful if you would like the bot to announce the fact it has started and granted access in a given team. The `ANNOUNCEMENT` environment variable supports a number of templating variables that will be instantiated based off of the current config. These are:

- `{USERNAME}` will be replaced with the username of the bot

- {CURRENT_TEAM} will be replaced with the team that the message is being sent in
- {TEAMS} will be replaced with the comma separated list of teams that the bot is running in

Examples:

```
export ANNOUNCEMENT="SSH CA bot starting up..."
export ANNOUNCEMENT="Hello! I'm {USERNAME} and I'm an SSH bot! See github.com/keybase/
↪bot-sshca for information on using Keybase for SSH."
export ANNOUNCEMENT="Hello! I'm {USERNAME} and I'm an SSH bot! I'm currently␣
↪listening in {TEAMS}."
export ANNOUNCEMENT="Hello! I'm {USERNAME} and I'm an SSH bot! Being in {CURRENT_TEAM}
↪ will grant you SSH access to certain servers. Reach out to @dworken for more␣
↪information."
```

3.1.8 Timeout

The KEYBASE_TIMEOUT environment specifies the number of seconds to wait for Keybase operations. If you are running the bot on an especially slow computer (ie a Raspberry Pi) or with a high latency internet connection, you may need to tune this. Defaults to 5 seconds.

Examples:

```
export KEYBASE_TIMEOUT="5"
export KEYBASE_TIMEOUT="15"
```

3.2 Developer Options

These environment variables are mainly useful for dev work. For security reasons, it is recommended always to run a production CA chat bot on an isolated machine. These options make it possible to run a CA chat bot on a machine where you currently are logged into another account.

Examples:

```
KEYBASE_HOME_DIR: /tmp/keybase/
KEYBASE_PAPERKEY: "paper key goes here"
KEYBASE_USERNAME: teamname-sshca-bot
```

CHAPTER 4

Best Practices

4.1 Teams and Channels

The SSH CA bot user needs to have write access in all of the teams used for granting SSH access in order for it to be able to store kssh client configs associated with each team. Since access to a team grants SSH access to servers, it is recommended to minimize the number of users with admin or owner permissions in the teams. Individual users of kssh only need to be given the read permission since they do not need to be able to edit or create files associated with a team.

It is also recommended to mute all notifications in the configured teams in order to minimize the number of notifications you get.

If you are using other bots in the same teams as the SSH CA bot (or if you wish to have normal conversation in those teams), you can use the `CHAT_CHANNEL` environment variable in order to configure a specific chat channel for all SSH CA messages.

4.2 Network Isolation

Due to the highly sensitive nature of the SSH CA bot, it is recommended to configure firewalls in order to block all access to the server running the CA bot. It is not recommended to use kssh to access the server of the CA bot itself in order to make it easier to respond to any outages.

4.3 Realms

There are two general approaches one can take when defining realms of servers. The first approach (described in the getting started directions) is to define realms for staging and production. This approach is useful for the common scenario where all developers should be given access to the staging environment but only certain people should be given access to production. The second approach is a more granular approach where you can define realms associated with teams. For example, one could have a realm of web servers, a realm of database servers, ... where a specific group of people is responsible for each class of server.

CHAPTER 5

Architecture

The Keybase SSH CA system works according to this diagram:

Note that this means that you do not need to modify your servers in any way or run any additional processes on your servers other than a standard OpenSSH daemon.

5.1 Network Architecture

Since all communication between the kssh client and the SSH CA server happens over Keybase chat, it is possible (and recommended) to firewall off the SSH CA server (where this bot is running) so it cannot be reached from the general internet. Additionally, note that the SSH servers that trust the SSH CA do not need to communicate with Keybase's servers or with the CA server and thus it is also possible to firewall off the SSH servers from the general internet. Clients running kssh need to have Keybase running locally with a connection to Keybase's servers.

CHAPTER 6

Troubleshooting

This file contains some general directions and thoughts on troubleshooting the code in this repo. This is not meant to be a comprehensive troubleshooting guide and is only a jumping off point.

6.1 `make generate` refuses to overwrite an existing key

In order to force `make generate` to overwrite the existing CA key (note that this will delete the existing CA key which means kssh will not work with any servers it currently works with), simply run:

```
FORCE_WRITE=true make generate
```

6.2 kssh is slow, but it works

When kssh starts, it has to search the KV store for every team you are in for the kssh config, which specifies the information that is needed in order to communicate with the CA chatbot. If you are only in a few teams, this is relatively fast but this can become much slower as the number of teams increases. You can avoid this search to reduce startup time by setting a default bot via `kssh --set-default-bot cabotname`.

6.3 kssh times out

If kssh times out with a message similar to:

```
Generating a new SSH key...
Requesting signature from the CA....
Failed to get a signed key from the CA: timed out while waiting for a response from␣
↪the CA
```

It means that for whatever reason, kssh is not receiving a response from the CA chatbot when it sends messages in Keybase chat. First, ensure that the CA chatbot is currently running. Next, attempt to determine what is happening by inspecting the chat messages inside of the teams configured with the chatbot. You should see a series of `Ack` and `AckRequest` messages going back and forth prior to a `Signature_Request:` and a `Signature_Response:` exchange. Ensure that you and the chatbot are in the correct teams such that they can read and respond to the messages. In addition, review the log output from the keybaseca chatbot. Note that it is required to run the keybaseca chatbot as a different user than you are using for kssh.

6.4 SSH rejects the connection

This likely means that you have not configured the SSH server correctly. Confirm that on the SSH server you are trying to access:

- `/etc/ssh/ca.pub` has an SSH public key in it
- `/etc/ssh/auth_principals/username-of-ssh-user` has the name of your Keybase team in it (or multiple comma separated keybase teams)
- `/etc/ssh/sshd_config` has the below two lines somewhere in it:

```
TrustedUserCAKeys /etc/ssh/ca.pub
AuthorizedPrincipalsFile /etc/ssh/auth_principals/%u
```

Also, ensure that these permissions are correctly set:

```
chmod 0645 /etc/ssh/auth_principals/
chmod 0644 /etc/ssh/auth_principals/*
chmod 0644 /etc/ssh/ca.pub
```

If that all looks good, review the getting started directions and ensure that you have followed the steps correctly. Additionally, it is recommended to compare your sshd_config file with the stock one for your OS to look for any non-standard config options. For example, setting `UsePAM no` may prevent the SSH CA from working. (*sshca.md* also has some additional information on how SSH CAs work that may be helpful). If you would like to follow an example, see the code in the `tests/` directory which contains integration tests (focus on Dockerfile-sshd for an example SSH server setup). If none of that works, the best strategy is to run SSH on the server on an alternate port and review the debug information. On the server run `/usr/sbin/sshd -dd -D -p 2222` and on the client run `kssh -p 2222 user@server` and inspect the debug logs.

6.5 Keybase is down

If Keybase is down, the bot will not work since it relies on Keybase chat for communication. In this scenario, you can manually sign SSH keys with the CA key. This can be done via `keybaseca sign --public-key /path/to/key.pub`. Alternatively, this can be done manually without relying on any of the tooling in this repository. To do so, place the CA private key in `~/cakey` and the CA public key in `~/cakey.pub`. Then run the command:

```
ssh-keygen \
  # The location of the ca key:
  -s ~/cakey  \
  # A unique ID for each key. Used to audit key usage
  -I unique-key-id \
  # The comma separated list of principals you wish to sign the key for. Eg "team.ssh.
↪prod,team.ssh.staging,team.ssh.root_everywhere"
  -n "team.ssh.prod,team.ssh.staging,team.ssh.root_everywhere" \
```

(continues on next page)

(continued from previous page)

```
  # How long the signature is valid for. +1d means one day. Valid units are `h` for␣
↪hour, `d` for day, `w` for week
  -V +1d \
  # Specify the password on the CA key (if exported via `keybaseca backup` there is␣
↪no password)
  -N "" \
  # The location of the public key you wish to sign
  /path/to/key.pub
```

You can then use the signed SSH key to SSH via `ssh -i /path/to/key.pub user@server`.

6.6 Default Users and kssh –provision

Default users are implemented using a custom SSH config file that inherits from the default one. This means that if you run:

```
kssh --set-default-user developer
kssh --provision
scp foo server:~/
```

It will not use the default user. There are two ways to work around this. If you do not need the default user to be kssh specific (eg if kssh is your primary way of accessing certain servers), then you can simply configure this default user globally by adding the below lines to `~/.ssh/config`

```
Host *
  User developer
```

If you do not want to do this, you can run scp with the kssh specific config file via:

```
scp -F ~/.ssh/kssh-config foo server:~/
```

Or analogously for rsync:

```
rsync -e "ssh -F $HOME/.ssh/kssh-config" foo server:~/
```

It may be useful to define aliases in your bashrc to simplify this:

```
alias kscp='kssh --provision && scp -F ~/.ssh/kssh-config'
alias krsync='kssh --provision && rsync -e "ssh -F $HOME/.ssh/kssh-config"'
```

6.7 Other

For any other issues, please open a Github issue or ping @dworken on Keybase! We want to make this project as reliable as possible so please let us know if there are any ways we can improve the project.

CHAPTER 7

Jumpboxes and Bastion Hosts

kssh should work correctly with jumpboxes and bastion hosts as long as they are configured to trust the SSH CA and the usernames are correct. For example:

```
kssh -J developer@jumpbox.example.com developer@server.internal
```

This can also be made easier by setting the kssh default ssh-username locally, then you won't have to specify it for each server.

```
kssh --set-default-user developer
kssh -J jumpbox.example.com server.internal
```

CHAPTER 8

OS Support

It is recommended to run the server component of this bot on linux and running it in other environments is untested. `kssh` is tested and works correctly on Linux, macOS, and Windows. If running on Windows, note that there is a dependency on the `ssh` binary being in the path. This can be installed in a number of different ways including Chocolatey or the built in version on modern versions of windows.

CHAPTER 9

Contributing and Additional Info

There are two separate binaries built from the code in this repo:

9.1 keybaseca

`keybaseca` is the CA server that exposes an interface through Keybase chat. This binary is meant to be run in a secure location.

```
NAME:
   keybaseca - An SSH CA built on top of Keybase

USAGE:
   keybaseca [global options] command [command options] [arguments...]

VERSION:
   0.0.1

COMMANDS:
     backup    Print the current CA private key to stdout for backup purposes
     generate  Generate a new CA key
     service   Start the CA service in the foreground
     help, h   Shows a list of commands or help for one command

GLOBAL OPTIONS:
   --help, -h     show help
   --version, -v  print the version
```

9.2 kssh

`kssh` is the replacement SSH binary. kssh handles provisioning (via the keybaseca-bot) new temporary SSH keys and is meant to be installed on each user's laptop.

```
NAME:
   kssh - A replacement ssh binary using Keybase SSH CA to provision SSH keys

USAGE:
   kssh [kssh options] [ssh arguments...]

VERSION:
   0.0.1

GLOBAL OPTIONS:
   --help                Show help
   -v                    Enable kssh and ssh debug logs
   --provision           Provision a new SSH key and add it to the ssh-agent. Useful␣
↪if you need to run another
                         program that uses SSH auth (eg scp, rsync, etc)
   --set-default-bot     Set the default bot to be used for kssh. Not necessary if␣
↪you are only in one team that
                         is using Keybase SSH CA
   --clear-default-bot   Clear the default bot
   --bot                 Specify a specific bot to be used for kssh. Not necessary if␣
↪you are only in one team that
                         is using Keybase SSH CA
   --set-default-user    Set the default SSH user to be used for kssh. Useful if you␣
↪use ssh configs that do not set
                                                a default SSH user
   --clear-default-user  Clear the default SSH user
   --set-keybase-binary  Run kssh with a specific keybase binary rather than␣
↪resolving via $PATH
```

9.3 Architecture

9.3.1 Config

Keybaseca is configured using environment variables (see docs/env.md for information on all of the options). When keybaseca starts, it writes a client config JSON blob to the KV store for each team in `$TEAMS`, at namespace `__sshca`, entry key `kssh_config`. This client config is how kssh determines which teams are using kssh and the needed information about the bot (eg the channel name, the name of the bot, etc). When keybaseca stops, it deletes all of the client configs.

kssh reads the client config in order to determine how to interact with a bot. kssh does not have any user controlled configuration. It does have one local config file stored in `~/.ssh/kssh-config.json` that is used to store a few settings for kssh. By default, this config file is not used. It is only created and meant to be interacted with via the `--set-default-bot`, `--clear-default-bot`, `--set-default-user`, `--clear-default-user` flags.

9.3.2 Communication

kssh and keybaseca communicate with each other over Keybase chat. If the `CHAT_CHANNEL` environment variable is specified in keybaseca's environment, keybaseca will only accept communication in the specified team and channel. This configuration is passed to kssh clients via the client configs stored in the KV store. If the `CHAT_CHANNEL` environment variable is not specified then keybaseca will accept messages in any channel of any team listed in the `TEAMS` environment variable. All communication happens via the Go chat bot library.

Prior to sending a `SignatureRequest`, kssh sends a series of `AckRequest` messages. An `AckRequest` message is sent until kssh receives an `Ack` from keybaseca. This is done in order to ensure that kssh has correctly connected to the chat channel and that the bot is responding to messages. Afterwards, a `SignatureRequest` packet is sent and keybaseca parses it and returns a signed key. Note that only public keys and signatures are sent over Keybase chat and private keys never leave the devices they were generated on.

9.3.3 SSH Operations

When the ssh-keygen command is available, ssh keys are generated via the ssh-keygen command. In this case, generated SSH keys are ed25519 keys. If the ssh-keygen command is not available, SSH keys are generated in pure go code and are ecdsa keys.

keybaseca uses the ssh-keygen binary in order to complete all key signing operations.

9.3.4 KBFS

keybaseca supports logging to a local or KBFS file. In order to ensure that keybaseca can run inside of docker (which does not support FUSE filesystems without adding the CAP_SYS_ADMIN permission), all KBFS interactions are done via `keybase fs ...` commands. This makes it so that keybaseca can run in unprivileged docker containers.

9.4 Integration Tests

This project contains integration tests that can be run via `./integrationTest.sh`. The integration tests depend on docker and docker-compose. The first time you run them, they will walk you through creating two new live keybase accounts to be used in the tests. The credentials for these accounts will be stored in `tests/env.sh`.

CHAPTER 10

SSH Certificate Authorities (CAs)

As described on our blog, SSH CAs are a way of building SSH authentication on top of a signing SSH keys. This document expands on how SSH CAs work, the different modes of operation, possible improvements, and other interesting things that can be done with CAs.

10.1 How an SSH CA Works

An SSH CA key is just a normal SSH key. This bot generates keys via:

```
ssh-keygen -t ed25519 -f ca-key -m PEM
```

A signature on a SSH key is a certificate that contains some additional information. A certificate contains:

1. The public key of the signed key
2. A key ID that can be used to identify who the key was issued to
3. A serial number that can be used manage revocation lists
4. A validity period where the key is considered valid only within that period of time
5. A list of principals

A list of principals on a certificate is simply strings that are shared for access control. The field is often used to encode roles a user has or groups that they are in, anything that helps the server decide whether or not to allow access (and how much access to grant). We use this field to include a list of keybase teams that the user is in. And the servers are configured to expect the same values (more below). As an example, here is the information encoded in a certificate created by this bot:

```
$ ssh-keygen -L -f  ~/.ssh/keybase-signed-key---cert.pub
/home/david/.ssh/keybase-signed-key---cert.pub:
        Type: ssh-ed25519-cert-v01@openssh.com user certificate
        Public key: ED25519-CERT SHA256:wdzTWhCrVeJrxRIC1KU5nJr8FbxxCUJt1IVeG7HYjmc
        Signing CA: ED25519 SHA256:OEhTm77qM7ZDwb5oltxt78FIpKraXCzxoaboi/KpNbM
        Key ID: "08a093ec-cb4e-4bc2-9800-825095418397:981b88e2-a214-4075-af77-
↪72da9600f34f"
```

(continues on next page)

(continued from previous page)

```
        Serial: 0
        Valid: from 2019-07-31T11:21:00 to 2019-07-31T12:22:50
        Principals:
                sshcademo.staging
                sshcademo.prod
                sshcademo.root_everywhere
                sshcademo.bot_access
        Critical Options: (none)
        Extensions:
                permit-X11-forwarding
                permit-agent-forwarding
                permit-port-forwarding
                permit-pty
                permit-user-rc
```

This certificate which was created by the CA bot is the bot's cryptographic assertion that I am in the specified teams. If I then attempt to SSH into a server with this certificate, it is the responsibility of the server to decide whether or not to allow the connection. This is done via adding two lines to `/etc/ssh/sshd_config`:

```
TrustedUserCAKeys /etc/ssh/ca.pub
AuthorizedPrincipalsFile /etc/ssh/auth_principals/%u
```

The first line specifies that `/etc/ssh/ca.pub` contains the CA public key (as generated by ssh-keygen above). The second line is how we define a mapping between the principals (for this bot: the Keybase teamnames) and the SSH users they are granted access to. For example, if we wanted this server to allow people with the principal `sshcademo.root_everywhere` to use the root user and people with the principal `sshcademo.staging` to use the "developer" user, we would create two files. First, `/etc/ssh/auth_principals/root` with contents `sshcademo.root_everywhere` and second `/etc/ssh/auth_principals/keybase` with contents `sshcademo.staging`.

When the SSH server accepts the connection, it will log the key ID and the serial number which can be combined with the CA bot's audit logs in order to track a connection to a specific keybase user.

```
Accepted publickey for developer from 65.202.161.38 port 56914 ssh2: ED25519-CERT ID␣
↪e662bf1e-0855-41e9-8951-87bf8c0b3614:f650a363-cd34-4ab0-b6bf-52faa120364d (serial␣
↪0) CA ED25519 SHA256:OEhTm77qM7ZDwb5oltxt78FIpKraXCzxoaboi/KpNbM
```

For more information on SSH CAs, here are a few more useful sources:

1. https://code.fb.com/security/scalable-and-secure-access-with-ssh/
2. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/deployment_guide/sec-using_openssh_certificate_authentication
3. https://medium.com/uber-security-privacy/introducing-the-uber-ssh-certificate-authority-4f840839c5cc

10.2 Future Improvements

Below are a few ideas for future improvements to this project. PRs welcome!

10.2.1 Key Encryption

Currently the CA key is stored on the filesystem unencrypted by the CA bot. As long as the CA bot is run on a well isolated machine, this is not seen as a significant security weakness. Nonetheless, this could be improved upon by

adding options that allow for encrypting the CA key.

10.2.2 Revocation

With an SSH CA it is possible to revoke signed SSH keys via revocation lists. A revocation list is a file on the server that contains a list of revoked keys. It is likely impractical to manually manage a revocation list, so this could be done via a cron job that periodically updates the revocation list. One of the main difficulties of designing this feature as part of the SSH CA bot is that a design goal of this bot was to not require running Keybase on every server. Thus, this revocation list would need to be somehow updated independent of Keybase.

10.3 Host Key Signing

SSH CAs can be used to sign SSH host keys. This would remove the below message and strengthen SSH by switching it away from a TOFU model.

```
$ ssh root@daviddworken.com
The authenticity of host 'daviddworken.com (2604:a880:400:d0::38c4:2001)' can't be␣
↪established.
ECDSA key fingerprint is SHA256:MmB6/g0vDrMkanuRc46n6JCDYPaPKHYsbDpLhQ3y1Yg.
Are you sure you want to continue connecting (yes/no)?
```

This feature has not been included because signing host keys is significantly different from signing user keys. Despite this, it could still be a useful feature to build on top of Keybase.