passtore.sh

Advanced pass management for teams

Cultural Commons Collecting Society SCE
mit beschränkter Haftung (C3S SCE)

Contents

Abstract

This script might be for you if you want to use pass for password management in a team that is so large that not everyone should be able to see everything. It is not a replacement for pass or QtPass, but an additional management tool, intended not to be used by all of the team but some designated password admins.

1 Background

Encrypting passwords using OpenPGP is a straight forward approach, especially if you use OpenPGP already in your team. We looked at `pass` and found it is a very nice tool for password management, so we examined how we could use it in our team. It stores passwords in separate files, using GnuPG for encryption.

By default, `pass` uses `~/.password-store` as the root directory for personal passwords and configures your OpenPGP encryption key in a file called `.gpg-id`. But you can also create subdirectories with different `.gpg-id` files to encrypt passwords for different OpenPGP keys. A `.gpg-id` file recursively defines the keys used, unless it is overruled by another `.gpg-id` file in a directory below.

You can even switch between multiple root directories, i. e., different password stores. In QtPass these are called *profiles*. An example `pass` password store could look like this:

```
store_root_dir/
    .gpg-id               # with key IDs of Alice and Bob
    alice/
        .gpg-id           # only key ID of Alice
        matrix.gpg        # ecrypted for Alice
        server1.gpg       # ecrypted for Alice
        server2.gpg       # ecrypted for Alice
    bob/
        .gpg-id           # only key ID of Bob
        mastodon.gpg      # ecrypted for Bob
        matrix.gpg        # ecrypted for Bob
    seafile/
        team_library.gpg  # ecrypted for Alice and Bob
```

In this example, the `team_library.gpg` was encrypted for both Alice and Bob because the `seafile` directory doesn't have it's own `.gpg-id` file, so `pass` uses the one from the nearest parent directory.

If you use `pass` to update the key list in a `.gpg-id` file, already encrypted passwords usually get re-encrypted reflecting the new key configuration.

1.1 The problem

As long as you use `pass` only for yourself, all is good as it is. But as soon as you need individual permissions for users to access password files, you need more control over the `.gpg-id` files in your directory structure. They are simple text files, so theoretically each user with write access could add any new keys to them and hope that during the next round of re-encryption they'll gain access to previously hidden passwords. In our example above, what if Bob, or anyone else for that matter, decided to silently add his key ID to `store_root_dir/alice/.gpg-id` and wait? As long as Alice doesn't notice, she'd expose all new or changed passwords to anyone with access to these additional OpenPGP keys.

`pass` already partially has a solution for that, in that it supports setting an environment variable defining a key that is used to *sign* `.gpg-id` files. If a signature is found, it will be validated before (re-)encryption. However, team members must actually have the environment variable set correctly, `pass` won't complain if it's missing ans silently accept any `.gpg-id` file.

1.2 How to solve this

What we're missing is a means to verify that

- all directories are supposed to exist
- each directory has it's own `.gpg-id` file so there's no accidental privilege escalation
- each `.gpg-id` file contains the intended key IDs (no more, no less)
- each `.gpg-id` file is validly signed with a defined key
- the encryption of each password file exactly matches the keys from its respective `.gpg-id` file

We also want to have easy access to the configuration, e. g.

- quickly list all OpenPGP keys or users that are known
- quickly show all keys/users that have access to given directories
- get an overview of all defined directories

2 Our implementation

The script `passtore.sh` provides these features. It uses

- a YAML configuration file to define
  - each user with email and OpenPGP key
  - groups of users, used for access control
  - all directories with standards for password generation and access groups
  - the keys used for signing
- as well as a local configuration file to define
  - at least one profile, i. e., the root directory of your password store, its YAML configuration file etc.
  - the same list of keys used for signing, once more

We call these (sub)directories »stores« because they all use their own `.gpg-id` file. In other contexts, their root directory is also sometimes referred to as »the password store«, please let this not confuse you.

Addressing the problem above, the one thing that must be ensured is the validity of signatures of the `.gpg-id` files, as well as of the YAML configuration. Since the YAML configuration sets the keys used for signing and should be included in the distributed password store, it was not enough to sign it, because an attacker could replace those keys and sign the config afterwards. That is why the local configuration has a copy of the signing key definition, and during its checks `passtore.sh` also ensures that both

are identical. An attacker would therefore also have to replace the key IDs on all machines that use the script in order to not get caught.

2.1 Read-write vs. read-only stores

To further limit the risk of an attack, `passtore.sh` is designed to support the use of two password store root directories. You can think of them as one being the actual password store (it is supposed to be read-only for everyone except password admins) and the other a turnstile for temporary use (with read-write access for everyone). The latter can be used by non-admins to provide new password files. `passtore.sh` can check their integrity and then move them to the read-only area. Both are expected to use the same directory structure as defined by the YAML file.

The fact that the actual password store can only be read but not changed by normal team members already limits the number of people who'd be able to compromise its integrity. Consequently, the team needs to appoint one or more members to do the task of maintaining the read-only store (»password admins«). They don't need read access to actual passwords, but they use `passtore.sh` to verify the integrity of the privilege structure the team has decided on.

If the idea of a globally writable password store immediately scares you, be aware that that's all you would have if you simply used `pass` for a team. Using an additional read-write root directory is not mandatory, though. Obviously, it comes with all the risks we're trying to avoid, if only for short periods of time and a somewhat higher probability of being noticed (see Remaining issues). You can use alternative channels to send password files to the password admins.

`passtore.sh` can also generate new passwords in a given store and encrypt them for the intended group of people, which could be done by a password admin directly so that passwords never exist outside the read-only area. This, on the other hand, might give that admin access to passwords during creation.

Note: `passtore.sh` itself does *not* enforce the read-write status of your directories! It is assumed that *you* make sure that the password store is read-only or read-write for parts of your team. There are various ways to do that (e. g., ACLs, read-only mounted file systems, read-only gitolite repositories, read-only Seafile libraries etc.). Choose a solution that fits your individual infrastructure.

2.2 Dependencies

The script needs these tools to work properly:

- `gpg` for OpenPGP stuff
- `pass` for password store management
- `yq` for YAML support

There's also two versions of the script available, one called `passtore.sh` and the other `passtore_static.sh`. The `passtore.sh` script was written using `bash_script_skeleton.sh` and therefore uses its function library dynamically. That is, if you want to use `passtore.sh` directly, a recent version of `bash_script_skeleton.sh` must be in your path. If you

don't want to install bash_script_skeleton.sh you can use passtore_static.sh instead of passtore.sh, which is the same script but with all the bash functions that it uses from bash_script_skeleton.sh statically copied into the file. It is therefore a much larger script and these functions will not get updated unless a new static version is being released. The static script might be more transparent for a code review.

2.3 Installation

If your system meets all the dependency requirements, simply copy passtore.sh or passtore_static.sh to a directory in your PATH.

Alternatively, you can clone this repository to be able to get updates and fixes, and use a symlink. E. g., if ~/bin is in your PATH:

```
# replace YOUR_DESIRED_LOCATION with the path you want to clone to
git clone https://github.com/C3S/passtore.git "${YOUR_DESIRED_LOCATION}"
chmod +x "${YOUR_DESIRED_LOCATION}"/*.sh
ln -s "${YOUR_DESIRED_LOCATION}"/*.sh ~/bin
```

To look for updates from time to time:

```
cd "${YOUR_DESIRED_LOCATION}"
git pull
```

2.4 Initial setup

Note: If you call passtore.sh (or passtore_static.sh, we'll only use passtore.sh from now on) without any arguments, it will show a usage message.

2.4.1 Configuration file

When passtore.sh is run for the first time, it initiates a local configuration file. Common for bash_script_skeleton.sh scripts, you can then call

```
passtore.sh --config
```

to open the configuration file for editing. It contains comments which hopefully explain what needs to be configured.

The configuration supports profiles, i. e., you can manage multiple password stores with passtore.sh, like one for your team and a second one just for you personally. Profiles are defined by several arrays containing key value pairs. Each array configures a different aspect, like the root directory of the read-only and read-write stores, the path to the YAML configuration, or the OpenPGP key IDs for signing. The keys in an array are always the names for a profile, they must therefore be consistent in all arrays. That is, if you want to use three profiles, each array must have exactly three key value pairs with the keys always using the same three names.

2.4.2 YAML configuration

The second configuration that needs to be set up is the YAML file. This needs to be done per profile. `passtore.sh` can initiate a commented example configuration file as a starting point. Just call

```
# replace PROFILE with a profile name from the local configuration file
passtore.sh -p "${PROFILE}" -y
```

to have it generated and opened for editing. After editing, don't forget to call

```
passtore.sh -p "${PROFILE}" -Y
```

(with a capital `Y`) to sign the YAML file.

It has a `keys` section (with subsections `users` and `groups`), a `defaults` section (currently only used for key generation settings), and a `stores` section to define password directories and who should have access.

Each section starts with a short description of what you should put there. Note that it is mandatory to keep the auto-generated group called `passadmin`, the example file also explains this in a comment: The OpenPGP keys of the `passadmin` members are used as the valid signing keys. Their key IDs must therefore match the signing keys in the local configuration file. So please adjust the members of this group, but you must not remove it.

Access to stores can only be given to groups, to simplify the script. If you want to grant access to individual users, create a group for each user as a workaround.

Note: If you keep the YAML configuration in the read-only store, non-admin users are able to run integrity checks on the stores, too. That's a good thing.

2.5 Workflow

`passtore.sh` reads the local configuration file to learn about defined profiles. When you select a profile (`-p` flag), the script knows which YAML configuration to use. Of the read-only (and read-write, if used) store, only the root directory must preexist and be writable by members of the `passadmin` group. They can create the correct directory structure with `passtore.sh`, which calls `pass init` with the particular configuration it fetches from the YAML file.

This means that before you touch anything in those root directories, you should always configure it in the YAML file *first*.

2.5.1 Example: A new password for the social media team

Let's assume you have a social media team which just registered a new Mastodon account. It now wants to store the login credentials in the read-only store, in a profile that is called `company`. If you were among the `passadmins`, your first task would be to add a new subdirectory to the YAML configuration. So you'd open an editor:

```
passtore.sh -p company -y
```

And then add something like this to the `stores` section:

```
- name: social media
  path: social_media
  groups:
    read:
      - social media

- name: mastodon
  path: social_media/mastodon
  groups:
    read:
      - social media
```

This defines both the `social_media` directory and its subdirectory `social_media/mastodon`, relative to the root directories. We set up both to ensure that each has a `.gpg-id` file so there's never confusion about who has access to what. You have given read access to a group called `social media` which must be defined among the `groups`, of course.

Now that you've changed the YAML file, you must sign it with your OpenPGP key, otherwise the next integrity check will indicate that someone tempered with the configuration:

```
passstore.sh -p company -Y
```

The next step would be to initiate the directories in both read-write and read-only root directories. This will also write the `.gpg-id` files and have you sign them:

```
passstore.sh -p company -s "social media" -i # read-only
passstore.sh -p company -s "social media" -w # read-write
passstore.sh -p company -s "mastodon" -i # read-only
passstore.sh -p company -s "mastodon" -w # read-write
```

The social media team can now use `pass` or similar to add its new password file to `social_media/mastodon` in the writable location. They should be aware that this is probably the most vulnerable step, as an attacker with access to the read-write directory with perfect timing could steal a password at this point (see Remaining issues below). So before encrypting their password, it seems advisable to double check the contents of the respective `.gpg-id` file.

Once that's finished, here comes the most interesting part, as you will now run an integrity check on the read-write directory:

```
passstore.sh -p company -R
```

If all went as expected, it runs without warnings or errors. But if, for example, the password file was encrypted for the wrong keys, you'll know now.

With all checks successfully completed, you can move the password file to the actual read-only location:

```
passstore.sh -p company -M "mastodon"
```

And finally, also run an integrity check in the full read-only directory:

```
passstore.sh -p company -C
```

3 Some final remarks

We actually didn't expect the script to become so large, otherwise we'd probably have gone for a different programming language. But for now it's what it is.

3.1 Remaining issues

As long as you can trust the `passadmin`s and there's no leaked or weak OpenPGP keys, we assume that `pass` combined with `passtore.sh` makes it possible to securely manage passwords in team settings. However, we can think of at least one weak point that this approach still has: There's kind of a race condition between a `passadmin` providing the `.gpg-id` and users using it for encryption. Here's what a potential attack could look like, if the attacker has access to the read-write directory:

1. Wait for the `passadmin` to provide a `.gpg-id` file
2. Replace it with an unsigned one that has the attackers key appended to it
3. Wait for the user to encrypt new passwords with it
4. Copy the encrypted password file
5. Restore the replaced `.gpg-id` and re-encrypt the password for only the intended users

The attacker now has a decryptable copy of the password, and if the `passadmin` first checks the integrity at this point, nothing dubious would come up. It is therefore crucial to either carefully check the `.gpg-id` before encrypting, or use a more secure channel for providing new passwords in the first place. The use of the writable location has the benefit of being directly usable for all team members with a `QtPass` profile or similar.

4 Contributing

To ask for help, report bugs, suggest feature improvements, or discuss the global development, please use the issue tracker.

4.1 Branches

Please note that all development happens in the `develop` branch. Pull requests against the `main` branch will be rejected, as it is reserved for the current stable release.

5 Licence

These scripts are distributed in the hope that they will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with these scripts. If not, see https://www.gnu.org/licenses/.