Secret Sharing

Setup & Usage

Date: 30. Aug. 2017

Table of Contents

1 Introduction....3
2 Setup....4
  2.1 Requirements....4
  2.2 Building the statically linked binaries....4
  2.3 Setting up the storage devices....4
  2.4 The air-gapped PC....6
3 Secret Sharing usage....7
  3.1 Verifying a storage device....7
  3.2 Generating a new key....7
  3.3 Splitting a key....7
  3.4 Merging a key....8
  3.5 Redistributing the key....8
4 Appendix A: Makefile usage....10
5 Appendix B: Technical implementation....11

1 Introduction

Using the Secret Sharing scripts presented here you can share a PGP keypair with three people. At least two are required to reconstruct and use the PGP key. This ensures availability whenever one team member leaves or a storage device (USB Stick) is lost. On the other hand, the information on one storage device is never enough to compromise the PGP keypair.

The setup consists of a set of scripts to perform common actions, a set of statically linked binaries which are used in the scripts and a set of procedures. This document will guide you through the setup of the scripts and binaries (chapter 2) and provide procedures on how to use them (chapter 3).

In this document it’s assumed the reader has a good understanding about PGP and how to use it at user level.

The technical implementation is described in Appendix B.

2 Setup

This chapter will guide you though the setup of everything you need to start with Secret Sharing.

2.1 Requirements

What you need:

- A PC with a Debian, Ubuntu or Gentoo installation for the build process
- 3 removable storage devices (USB Sticks) with a capacity of at least 64 MB
- Optionally an air-gapped PC with all wireless/networking disabled
- The scripts and Makefile published here: <url>

First, you'll need to build the statically linked binaries (described in chapter 2.2). After that you can set up the storage devices (described in chapter 2.3). Setting up the air-gapped PC is described in chapter 2.4.

2.2 Building the statically linked binaries

In this chapter we'll build the statically linked binaries. You might wonder why we need them in the first place. The answer is: To be able to run the Secret Sharing scripts with the least possible dependencies on the system you run them on. Firstly this is because you want them to Just Work[tm] without the need to install additional software to allow them to work. The second reason is one of trust. Do you actually trust the binaries installed on the system you are using ? By building our own binaries we can have maximum trust in the binaries we use and only depend on the kernel running on the PC where we use the Secret Sharing script.

To build the binaries you'll need a recent Debian, Ubuntu or Gentoo installation with either gcc5 or gcc6. Other distributions may work but are untested. Gentoo users are, because of the source-based nature of Gentoo, ready to go without installing additional software. Debian and Ubuntu users need to install the build-essentials and gawk packages.

The complete build process takes about 1,5 GB of disk space, if your system has 4 GB or more memory available I recommend using a tmpfs. You can mount it anywhere, and I recommend using -o size=2G,exec to make sure there is enough space available and execution of files is permitted. The complete mount command would be:

```
mount -t tmpfs tmpfs <your mountpoint> -o size=2G,exec
```

Now, copy the Makefile to the location where you wish to start the build process and cd into that location. Now simply type make to start the process. On recent hardware you can expect it to run for about 15-30 minutes. The result should be a directory called static_bin with the following files: base64, bash, cat, dd, df, gpg2, gpg-agent, grep, kill, mkdir, mv, openssl, openssl.cnf, pinentry-tty, rm, sha512sum, split, unzip and zip.

For more information on using the Makefile see Appendix A.

2.3 Setting up the storage devices

Before we start copying files to the storage devices, we first create a template of what we want to copy, so installation will be limited to a single copy command.

First, create a new, empty directory, this will contain the files we need to copy to the storage devices later. Within this directory create a “secret_sharing” and a “data” directory. Now, copy the provided scripts (split_key.sh, start_gpg.sh, verify_files.sh, merge_key.sh and generate_key.sh) into the secret_sharing directory. Within the secret_sharing directory create a “work” directory, we will leave this empty to be used as a mountpoint for a tmpfs. Now create a “static_bin” directory under the “secret_sharing” directory, and copy the static_bin files generated by the Makefile into this directory.

The result should be a file listing like this:

```
./secret_sharing
./secret_sharing/static_bin
./secret_sharing/static_bin/base64
./secret_sharing/static_bin/bash
./secret_sharing/static_bin/cat
./secret_sharing/static_bin/dd
./secret_sharing/static_bin/df
./secret_sharing/static_bin/gpg2
./secret_sharing/static_bin/gpg-agent
./secret_sharing/static_bin/grep
./secret_sharing/static_bin/kill
./secret_sharing/static_bin/mkdir
./secret_sharing/static_bin/mv
./secret_sharing/static_bin/openssl
./secret_sharing/static_bin/openssl.cnf
./secret_sharing/static_bin/pinentry-tty
./secret_sharing/static_bin/rm
./secret_sharing/static_bin/sha512sum
./secret_sharing/static_bin/split
./secret_sharing/static_bin/unzip
./secret_sharing/static_bin/zip
./secret_sharing/work
./secret_sharing/generate_key.sh
./secret_sharing/merge_key.sh
./secret_sharing/split_key.sh
./secret_sharing/start_gpg.sh
./secret_sharing/verify_files.sh
./data
```

The verify_files.sh is a script to verify the integrity of all files. On order to perform this task we need to perform some extra steps. Cd into the secret_sharing directory and execute the following command:

```
sha512sum split_key.sh start_gpg.sh merge_key.sh generate_key.sh \
     static_bin/* > verify_files.sha512
```

This will generate the verify_files.sha512 file which verify_files.sh uses.

Besides all other files, verify_files.sh also verifies verify_files.sha512 and itself.
Verify_files.sha512 is verified by comparing the hash to a stored value, verifying itself is a bit more tricky. Of course we can’t just calculate the hash ans store it in the script, because that will change the script, and thus the hash. To solve this Catch-22 situation the script filters out all lines that end with #HASHLINE. This is a comment to bash, so it won’t affect it’s function, and allows the script

to skip the line where the hash is stored so the hash doesn't change when the hash itself is updated. As an extra check the script displays an error if the number of #HASHLINE lines is different than 1.

First we'll have to make sure the hash for verify_files.sha512 is correct. Run

```
sha512sum verify_files.sha512
```

and store tha hash value. Edit the verify_files.sh script and go to line number 61. This is where the hash for verify_files.sha512 is stored. Now go to line 46 and **insert a new line** with the content:

```
echo $MYHASH #HASHLINE
```

This will echo the correct hash value for verify_files.sh, and the #HASHLNIE tag will make sure the hash doesn't change when we remove this line later on. Now exit the editor and run the ./verify_files.sh script.

The result should be an "OK" result for all files except verify_files.sh itself. The correct hash is printed right above that result. Copy that value and open the verify_files.sh file again with an editor. Remove the extra line you just added and after that modify the hash on line 47 with the value you just copied. Exit the editor, and run ./verify_files.sh again, and all files should be OK.

As an extra step you can run sha512sum on verify_files.sh and verify_files.sha512 and store the result on a wiki or in password manager. This will enable you to have an external verification of the files on a storage device.

The last step is to copy the files we have to all three removable storage devices. First prepare them. In my case I first wiped them and formatted them with a Linux-friendly file system (ext2,3,4). Now copy the secret_sharing and data directory onto the root of the storage devices.

As a double-check you can run the verify_files.sh script on every storage device, which should end with all OK results.

2.4 The air-gapped PC

To make the setup as effective as possible in protecting the PGP keypair, it is best to use them on an air-gapped PC only. As a starting point it's best to have a PC with the least amount of wireless connectivity available and preferably has 3 or more USB ports to connect the storage devices.

If the PC has any wireless connectivity, check out its BIOS or UEFI for possibilities to disable them.

For software, I used a Gentoo Minimal Installation LiveCD image written to the local disk. This has the advantage that it takes quite some effort to make permanent changes, and with it being a minimal CD, the environment is as bare as possible.

3 Secret Sharing usage

The next chapters will describe actions you may want to do with the PGP key protected by Secret Sharing and work-instruction level. The general idea behind it is that the key NEVER leaves the tmpfs on the air-gapped PC, so any action that has to be done on other data (actual files or keys that have to be signed) has to be transferred offline to the air-gapped PC.

3.1 Verifying a storage device

Before using any of the other scripts you need to make sure all scripts and binaries are exactly the same as when they were placed on the storage device. For this purpose we can use the verify_file.sh script. If you stored the sha512sum of verify_files.sh and verify_files.sha512 somewhere, it's good to check those first. Assuming you mounted the device at /mnt/sda1 change to the directory /mnt/sda1/secret_sharing and run the command: sha512sum verify_files.sh verify_files.sha512

If these hashes match your stored values, run the verify_files.sh script itself by simply starting: ./verify_files.sh. Again, all checks should result in an OK result.

Once that is completed you can continue to do whatever you wanted to do.

```
cd /mnt/sda1/secret_sharing
sha12sum verify_files.sh verify_files.sha512
./verify_files.sh
```

3.2 Generating a new key

For this procedure you'll need 1 storage device because it will only generate a new key. Of course you'll need all 3 of them when you'll split it. In this example I assume the storage device to be mounted at /mnt/sda1.

- Change to the directory /mnt/sda1/secret_sharing.
- Verify the contents using the procedure "Verifying a storage device".
- Start ./generate_key.sh, this time it will quit immediately and give you a command to mount a tmpfs at the work directory. Execute this command.
- Start ./generate_key.sh, this time it will start gpg2 to generate the new key.
- The passphrase can be stored in a password manager.

Warning: The key is stored in the tmfs only and will be lost on shutdown, reboot of the PC or unmount of the tmpfs.

3.3 Splitting the key

To split the key you'll need all 3 storage devices. This procedure assumes a key is already available, either because it was generated or because it was merged. It also assumes the storage device you are working on is mounted at /mnt/sda1.

- If you haven’t done it before, verify the scripts and binaries using the procedure “Verifying a storage device”.
- Go to the directory /mnt/sda1/secret_sharing and start ./split_key.sh. It requires one argument as a reference to the key you want to split. This can be an e-mail address, a (part of) the name, or the key id.
- The last step in the split process is to encrypt all parts with a personal AES passphrase. Let the owners of the three storage devices each choose their own passphrase and store it securely.
- The result will be three files stored at /mnt/sda1/secret_sharing/work called set1.zip.aes, set2.zip.aes and set3.zip.aes.
- Copy set1.zip.aes to the data directory at storage device 1.
- Copy set2.zip.aes to the data directory at storage device 2.
- Copy set3.zip.aes to the data directory at storage device 3.

3.4 Merging the key

For this procedure you’ll need at least 2 storage devices. Choose one to work with, this example assumes that storage device to be used is mounted at /mnt/sda1.

- Verify the scripts and binaries using the procedure “Verifying a storage device”.
- Go the to /mnt/sda1/secret_sharing directory and start ./merge_key.sh, this script won’t do anything right now, but it’ll give you a command to mount a tmpfs at the work directory. Execute that command as root.
- Copy the file set1.zip.aes from storage device 1 to /mnt/sda1/secret_sharing/work (if available).
- Copy the file set2.zip.aes from storage device 1 to /mnt/sda1/secret_sharing/work (if available).
- Copy the file set3.zip.aes from storage device 1 to /mnt/sda1/secret_sharing/work (if available).
- Start the ./merge_key.sh script again. It will start with the decryption of the set files, the passphrases are personal to the owner of the file/storage device.
- If enough parts are decrypted successfully the resulting key will be imported to the keyring.

3.5 Redistributing the key

In case one set got lost or damaged for whatever reason, you can follow this procedure to re-create three parts again. Keep in mind that all three sets will change, so you’ll need to update all three storage devices at the end of this procedure.

- If you start using a new storage device, prepare it using the “Setting up the storage devices” procedures
- Use the procedure “Merging a key” to reconstruct the key
- Use the procedure “Splitting a key” to split the key again
- Make sure to copy all three parts to their storage devices because all three parts have changed

3.6 Performing other actions with the key

Of course the key is only useful if you can use it to perform some actions like signing or de/encrypting.

- Fist you’ll need to merge the key using the “Merging the key” procedure on the air-gapped PC
- Copy the file you need to work on to the air-gapped PC at /mnt/sda1/secret_sharing/work
- Do whatever you need to do, use the ./start_gpg.sh script instead of the gpg command. This ensures the gpg-agent is running and the keyring on the tmpfs is used.
- Copy the result of your work away from the air-gapped PC, and verify successful execution of your action.
- When ready, unmount the tmpfs and storage devices
- Shutdown the air-gapped PC

4 Appendix A: Makefile usage

The Makefile supplied to build the statically linked binaries can be used in several other ways than just issuing “make”. When experimenting, watch out for shark attacks. These are the supported possibilities:

- make downloads
  Will download all required downloads and verify them. This can be used to complete the build process on an offline PC.
- make patches
  The Makefile automatically applies some patches, mainly to other Makefiles to enable static linking. This command can be used to write the patches (without applying them) to review them.
- make base64|bash|cat|dd|df|gpg2|gpg-agent|grep|kill|mkdir|mv|openssl|openssl.cnf|pinentry-tty|rm|sha512sum|split|unzip|zip
  If you supply the name of a binary to build, it will only build that binary. So “make base64” will result in only the base64 binary in the static_bin directory. Dependencies are automatically taken into account.
- make coreutils
  This will build all binaries which are supplied by the coretutils package (base64, cat, dd, df, kill, mkdir, mv, rm, sha512sum, split).

5 Appendix B: Technical implementation

The split process consists of the following steps:

- Export of the public and private key from the GPG database
- Generating a new random AES key
- Encrypt the public and private key with the generated AES key
- Split the encrypted public and private key and AES key in three parts
- Generate hashes of all files generated so far
- Create 3 ZIP files, each containing 2 different parts of the public, private and AES key and the hashes.
- Encrypt the ZIP files using personal passphrases

The merge process consists of the following steps:

- Decrypt the encrypted ZIP files using the personal AES passphrases
- Unzip all available ZIP files
- Merge the encrypted public and private key and the AES key
- Decrypt the public and private keys with the AES key
- Check the hashes for all files
- Import the public and private key in the GPG database