Search Medium

Published in Level Up Coding

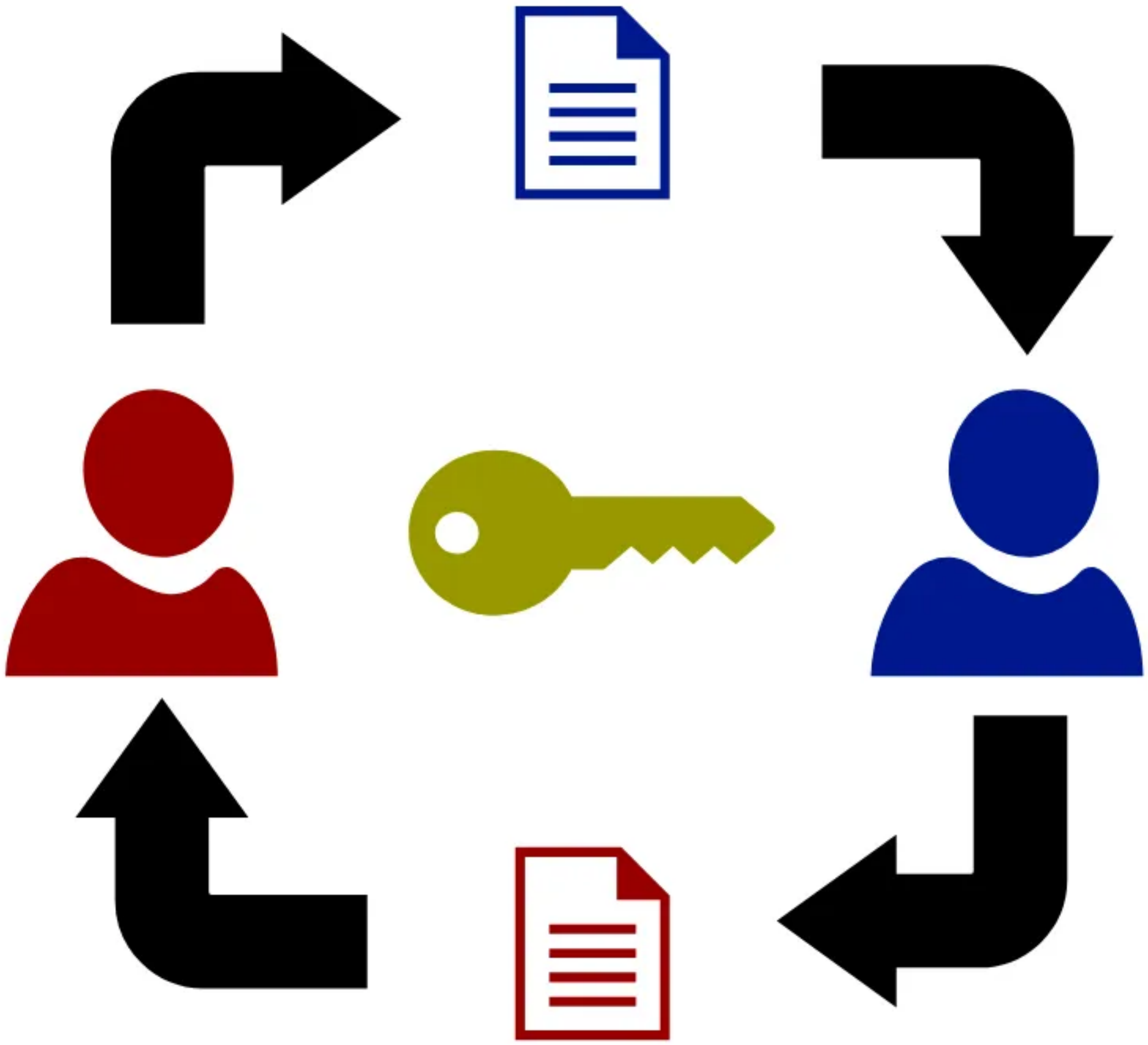
This is your **last** free member-only story this month. Sign up for Medium and get an extra one

Richard Fan Follow

Mar 19, 2020 · 6 min read · · Listen

Save

Using SOPS and git hooks to share secrets — Part 2



What we have done so far

In part 1, we have set up our repository to use Mozilla SOPS to encrypt secret files before commit. But the encryption/decryption process still relies on human interaction, which is not a good practice in DevOps. In part 2, we are going to automate this process using githooks.

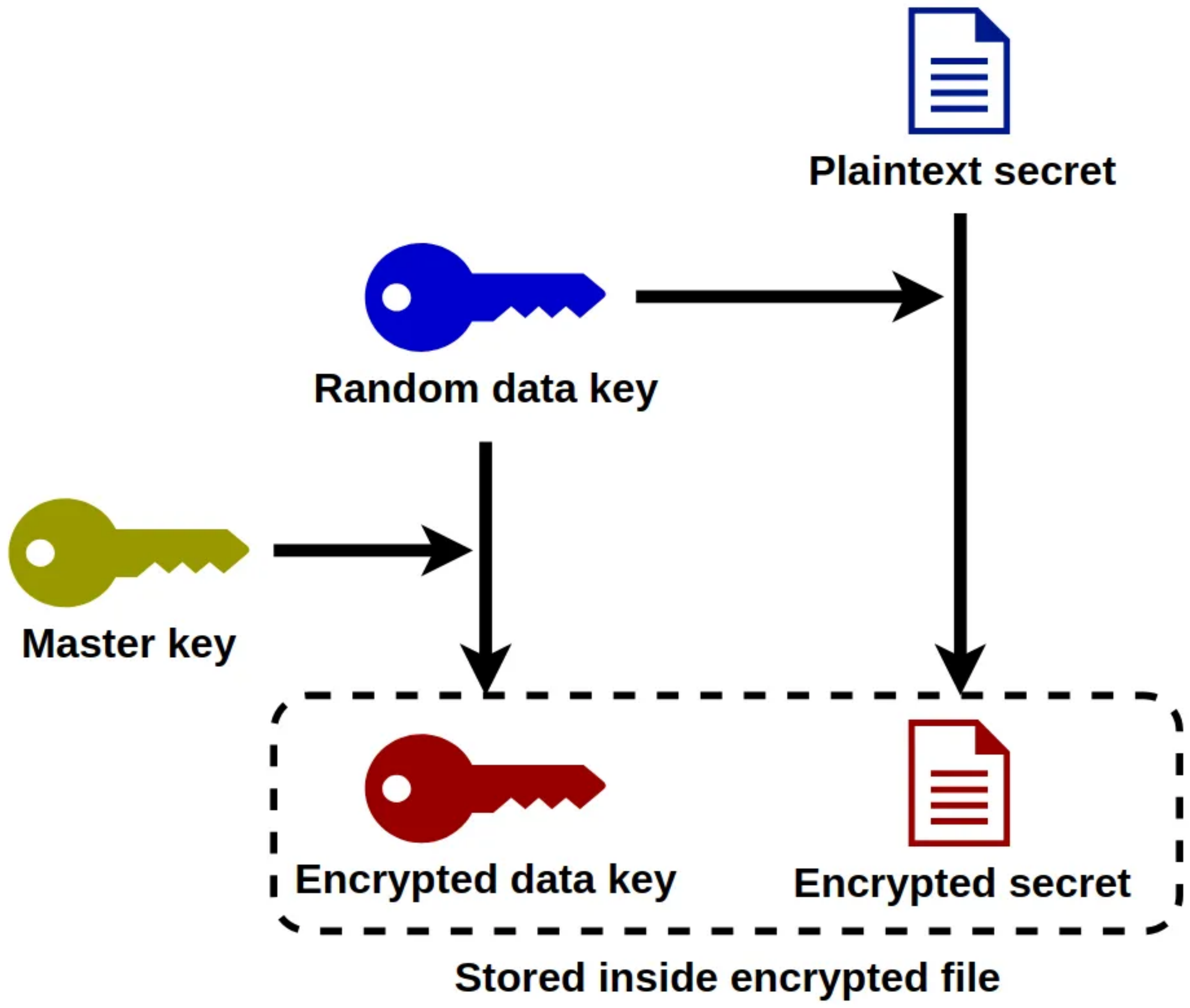
What is githooks

githooks is a set of shell files that will be run at different stages of git actions, allowing us to customise our development workflow. In this session, we will use 3 hooks: `pre-commit`, `post-merge` and `post-rewrite`. You can find my code in my GitHub repository.

Problems we have to solve

Problem 1: Cannot track if files changed

SOPS uses envelope encryption, our data is not directly encrypted by the key (master key) we provide (in our case, it's the KMS customer key). Instead, they are encrypted by randomly generated keys (data key), the master key then encrypts the data key. The encrypted data key will then be stored as the metadata inside the encrypted file. Because the data key is random, if we encrypt the same unmodified file, we still get different results every time.



SOPS uses random data key to encrypt files

If we naively encrypt every file before each commit, we will end up changing it every time although we are actually not. Detecting genuine file changes is the first problem we have to solve

Problem 2: Unable to observe conflict

Because we are not storing plaintext files inside the repository, when conflict occurs, Git can only give you the diff view of the encrypted files. But our brain is not a decrypter, we have to see how the conflict occurs in the plaintext version. This is the second problem we have to solve.

How to use the hooks

githooks are shell files that will be run on local machines. For security reason, these files cannot be cloned from remote repositories. Otherwise, hackers can inject malicious code to computers using their repositories.

To use the hooks, we need to manually copy the hook files under `.git/hooks` folder in your cloned repository.

Let's dive into the code

1. secret_files

To let my hooks know where our secret files locate, I have created the `.secret_files` file at the root level. It's just a simple text file, listing out the relative path of all the files we want to encrypt.

2. pre-commit

`pre-commit` is triggered before commit actually happens, I use this hook to do the data encryption.

```
# Get all the filenames of all secret files
for originFile in $(cat .secret_files); do

        # Generate encrypted filename "<filename>.enc.<extension>"
        filename="${originFile%%.*}"
        ext="${originFile#*.}"
        encryptedFilename="$filename.enc.$ext"
```

The hook first gets the list of the secret files. For each file, it will generate the corresponding filename for the encrypted version (adding `.enc` before the file extension).

```
        # If encrypted file not exist, generate one
        if [[ ! -e $encryptedFilename ]];then
                sops -e --output $encryptedFilename $originFile
                echo "$originFile modified, please commit again"
                anyFileChanged=1
                continue
        fi
```

Then, it will check if the encrypted version exists. If not, it will create one by encrypting the plaintext, I will talk about the `anyFileChanged` flag later.

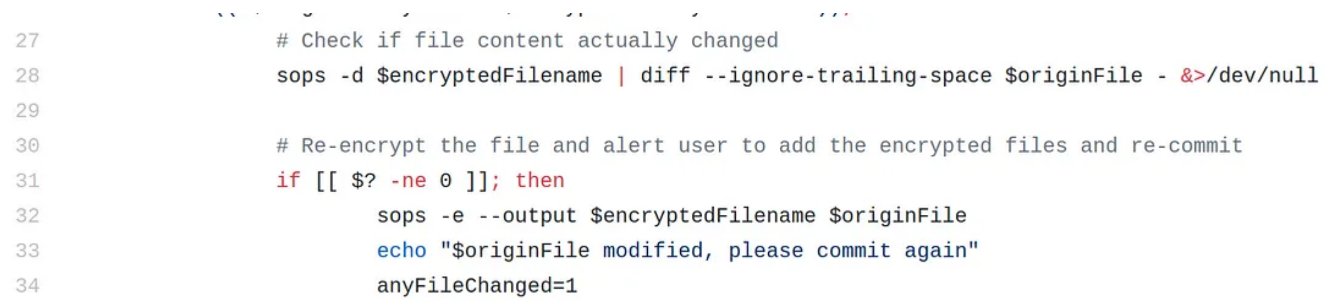
```
        # Get files last update time
        originModifyTime=$(date -r $originFile +%s)
        encryptedModifyTime=$(date -r $encryptedFilename +%s)

        # Check if origin file changed after encrypted one
        if (( $originModifyTime - $encryptedModifyTime > 0 ));then
```

If the encrypted file exists, it will continue comparing the last update time of the plaintext and encrypted files.

I do so because if the encrypted version is more updated than the plaintext version, I can assume the encrypted one is up-to-date. I can then skip the actual content comparison to lower key usage. (Data encryption is expensive, both by cost and computer resource)

Of course, if the user directly edits the encrypted file, it may be the problem.

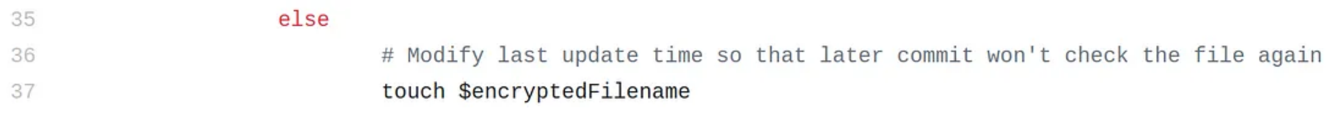


```
                # Check if file content actually changed
                sops -d $encryptedFilename | diff --ignore-trailing-space $originFile - &>/dev/null

                # Re-encrypt the file and alert user to add the encrypted files and re-commit
                if [[ $? -ne 0 ]]; then
                        sops -e --output $encryptedFilename $originFile
                        echo "$originFile modified, please commit again"
                        anyFileChanged=1
```

If the plaintext version has been updated after the encrypted version, the hook will check if their contents are actually different. I do it by decrypting the encrypted version and use `diff` to compare content.

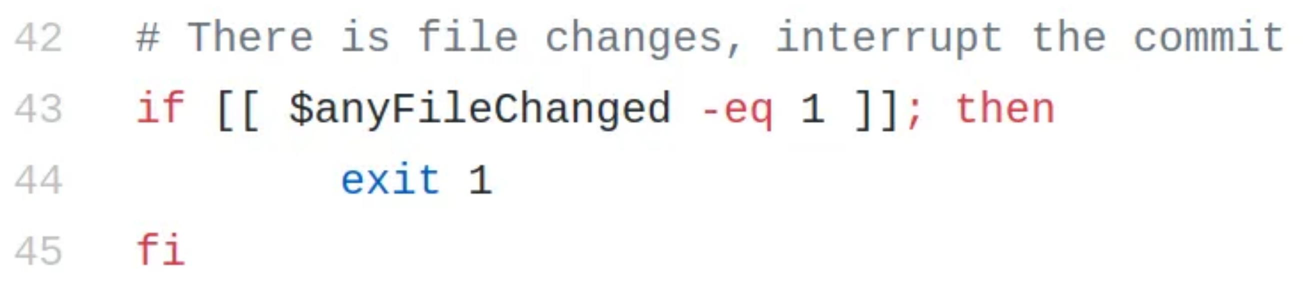
If they are different, it will encrypt the plaintext version and replace the encrypted version. Note that I always treat the plaintext version as the truth because developers are more aware of the plaintext than the cypher text.



```
                else
                        # Modify last update time so that later commit won't check the file again
                        touch $encryptedFilename
```

If the content is actually the same, it won't encrypt the file. It solves our first problem.

However, the hook still modifies the last update time of the encrypted file. It's to prevent the hook from comparing the unmodified files again and again during later commits.



```
# There is file changes, interrupt the commit
if [[ $anyFileChanged -eq 1 ]]; then
        exit 1
fi
```

The last thing is the `anyFileChanged` flag. If there is any secret file that has been changed, the user will be notified and the commit will be terminated by the non-zero exit code.

The user can now stage the newly encrypted file and do the commit again.

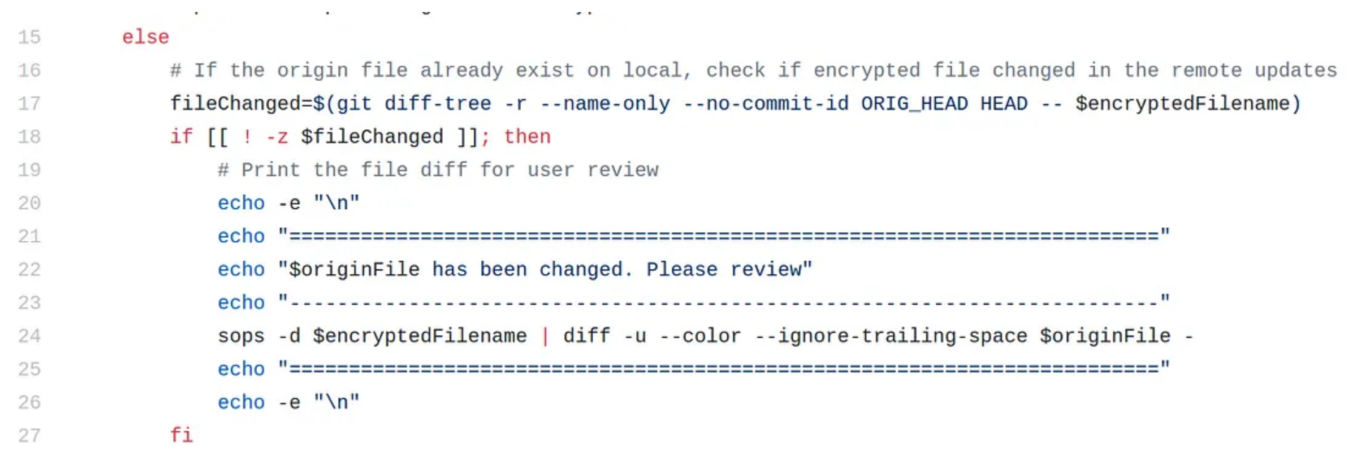
3. post-merge and post-rewrite

These 2 hooks are actually doing the same thing. The difference between them is that `post-merge` will be triggered after `git merge` while `post-rewrite` will be triggered after `git rebase`. Developers usually pull remote update by merging or rebasing, so I included the decryption process into these 2 hooks.

Note that these 2 hooks are triggered after the actions. So the remote files are already updated to the local.

`post-rewrite` is actually calling `post-merge` file, so we can simply go into `post-merge`.

Like `pre-commit`, the hook first loop on the secret file list and generate the corresponding filename for encrypted version. Then, it checks if the plaintext version exists. If not, it creates one by decrypting the encrypted version.



```
    else
        # If the origin file already exist on local, check if encrypted file changed in the remote updates
        fileChanged=$(git diff-tree -r --name-only --no-commit-id ORIG_HEAD HEAD -- $encryptedFilename)
        if [[ ! -z $fileChanged ]]; then
            # Print the file diff for user review
            echo -e "\n"
            echo "========================================================================="
            echo "$originFile has been changed. Please review"
            echo "-------------------------------------------------------------------------"
            sops -d $encryptedFilename | diff -u --color --ignore-trailing-space $originFile -
            echo "========================================================================="
            echo -e "\n"
        fi
```

The main part of the hook happens if the plaintext version exists. First, the hook will check if the secret file included inside the current pull. I do that by running `git diff-tree` on `HEAD` (the commit we are currently at) and `ORIG_HEAD` (the commit right before merge/rebase) and see if the files are inside this diff.

If the file has been changed, the hook will perform `diff` on the current plaintext version and the newly updated file and show the result to the user.

As I have said, I always treat the plaintext version as the truth, so the hook won't directly replace the local version. It prompts the user how the file has been changed and let the user decide how to merge 2 versions.

Limitation

1. Only works on CLI

As you see, the hooks rely on `echo` to prompt users on file changes. While it works on CLI, it is not guaranteed that GUI git client will redirect those `echo` output to the user.

I have tested on VS Code, it works on the `pre-commit` hook. It properly shows the "$origin File modified, please commit again" message on alert box and stop the commit. But on `post-merge` hook, it silently finishes the pull action although my secret file has been changed on the remote tree.

2. User may ignore the conflict prompts

On `post-merge` hook, I prompt users to resolve conflicts. However, it is not enforced like how git works, users may ignore the conflicts and commit their own version to the remote tree and overwrite their teammates' works.

3. Git actions are complicated

The hooks may work on most common cases. However, Git has many different features like `stash`, `revert`, `squash`, etc. Combining all of the action, there are many combinations, it is difficult to test how these hooks act in those cases.