CHEAT SHEETS

Rewriting your git history, removing files permanently [cheat sheet included]

Learn how to safely remove confidential information from your git repository. Whether you need to excise an entire file or edit a file without removing it, this tutorial will guide you through the process. Plus, get tips on preventing future headaches with GitGuardian!

GUEST EXPERT
29 JAN 2021 · 10 MIN READ

Share

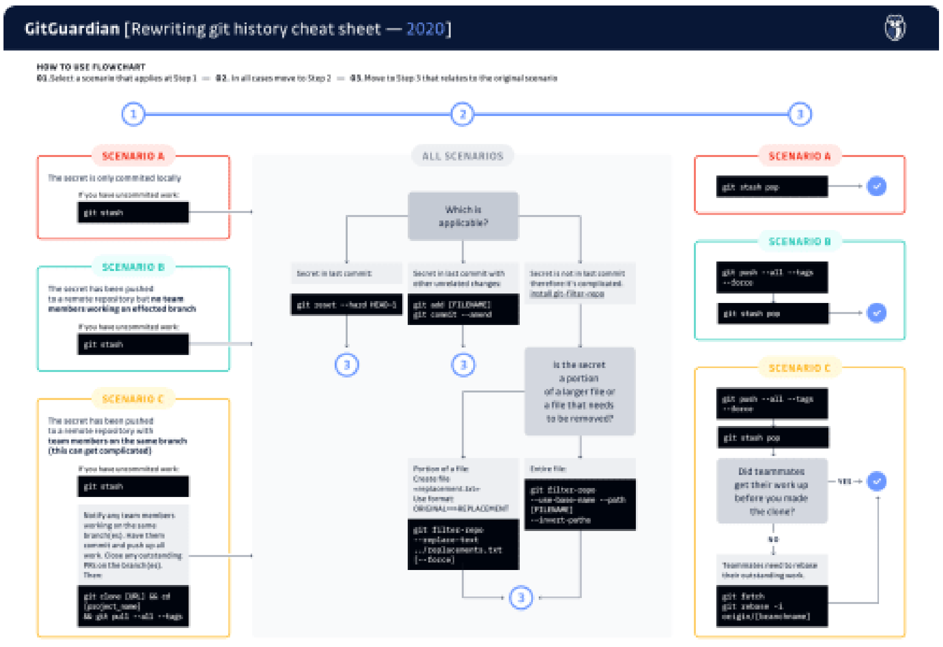
Table of contents

You know that adding secrets to your git repository (even a private one) is a bad idea, because doing so **risks exposing confidential information to the world**. But mistakes were made, and now you need to figure out how to excise confidential information from your repo. **Because git keeps a history of *everything*, it's not often enough to simply remove the secret or file, commit, and push**: we might need to do a bit of deep cleaning.

Thankfully, for simpler cases, git provides commands that make cleaning things up easy. And in more complicated cases, we can use git-filter-repo, a tool recommended by the core git developers for deep cleaning an entire repository.

depending on what the secret protects. If you don't know how to revoke it, you will need help from the owner of the resource protected by the secret.

> Need to quickly see what scenario applies to you?
> check out our cheatsheet flow chart below


GitGuardian [Rewriting git history cheat sheet — 2020]
HOW TO USE FLOWCHART
01. Select a scenario that applies at Step 1 — 02. In all cases move to Step 2 — 03. Move to Step 3 that relates to the original scenario
1
2
3
SCENARIO A
The secret is only commited locally
If you have uncommited work:
SCENARIO B
The secret has been pushed to a remote repository but no team members working on effected branch
If you have uncommited work:
SCENARIO C
The secret has been pushed to a remote repository with team members on the same branch (this can get complicated)
If you have uncommited work:
Notify any team members working on the same branch(es). Have them commit and push up all work. Close any outstanding PRs on the branch(es). Then:
ALL SCENARIOS
Which is applicable?
Secret in last commit
Secret in last commit with other unrelated changes
Secret is not in last commit therefore it's complicated. Install git-filter-repo
Is the secret a portion of a larger file or a file that needs to be removed?
Portion of a file: Create file <replacement.txt> Use format: ORIGINAL==>REPLACEMENT
Entire file:
Did teammates get their work up before you made the clone?
YES
NO
Teammates need to rebase their outstanding work

```
git stash
git clone [URL] && cd [project_name] && git pull --all --tags
git reset --hard HEAD~1
git add [FILENAME]
git commit --amend
git filter-repo --replace-text ../replacements.txt [--force]
git filter-repo --use-base-name --path [FILENAME] --invert-paths
git stash pop
git push --all --tags --force
git fetch
git rebase -i origin/[branchname]
```

Download the git history cheatsheet

Now, let's consider different scenarios to see how to clean things up.

Have you pushed your work up yet?

NO

No? That's great. Please don't push it up just yet.If you have any uncommited work, we can use `git stash` to save it. This sets your work aside in a temporary "stash" so that we can work with the git repository without losing anything you haven't committed yet. When we're done cleaning things up, you can use `git stash pop` to restore your work.

your existing history, things get more complicated if there are other people working on this branch. If you work alone, there's nothing to do at this point, you can skip to the next step. If you work as part of a team, things get more complicated because we need everyone to act in a coordinated way.

First of all, we need to determine who else is affected by the secret's presence, because we'll need to coordinate everyone's actions. If the secret only appears in the branch you're working on, you only need to coordinate with anyone else who is always working off of that branch. However, if you found the secret lurking further back in git history, perhaps in your `master` or `main` branch, you'll need to coordinate with everyone working in the repository.

Let the others affected know that a secret was found that needs to be excised from everyone's git history. When you edit the git history to remove a file, it can cause problems with your teammates' local clones; moreover, they can end up re-inserting the secret back into the public repository when they push their work. So it is important that everyone affected is in sync for the excision to work. This means that everyone needs to stop what they are doing, close outstanding PRs, and push up work that's in progress.

Now, let's make a fresh clone.

- Delete your existing clone in its entirety.
- Make a fresh clone with `git clone [repository URL]`
- Change into the project directory with `cd [project name]`
- Download the entire repository history: `git pull --all --tags`.

The last step will look a little bit familiar. `git pull` tells git to grab updates from the remote repository, and apply them in the current branch (when it makes sense to do so, that is, when the local branch is set to track a remote branch). But git is smart, it doesn't pull everything down, only what's needed. This is where the `--all` flag comes in. This flag tells git to grab every branch from the remote repository. And the `--tags` flag tells git to grab every tag as well. So this command tells git to download the entire repository history, a complete and total clone. We have to do this, because it is possible that the commit containing the secret exists in more than one branch or tag. We need to ensure

Move on to the next step.

How complicated is the situation?

The secret is in the last commit, and there's nothing else in the last commit

In this case, we can drop the last commit in its entirety. We do this with

```
git reset --hard HEAD~1
```

What does this command do? Let's break it apart a little bit. `git reset` is how we tell git that we want to undo recent changes. Normally, this command by itself tells git to unstage anything we've added with `git add`, but haven't committed yet. This version of resetting isn't sufficient for our purposes. But `git reset` is more flexible than that. We can tell git to take us back in time to a previous commit as well.We do that by telling git which commit to rewind to. We can use a commit's identified (it's "SHA"), or we can use an indirect reference. `HEAD` is what git calls the most recent commit on the checked out branch. `HEAD~1` means "the first commit prior to the most recent" (likewise `HEAD~2` means "two commits prior to the most recent").
Finally, the `--hard` tells git to throw away any differences between the current state and the state we're resetting to. If you leave off the `--hard` your changes, including the secret, won't be discarded. With `--hard`, the differences will be deleted, gone forever (which is precisely what we want!).

Once you've done a hard reset, that's it! You're done. Your work has been destructively undone, and you can pick back up where you were.

The secret is in the last commit, but there were other changes too

usual. Then, instead of making a new commit, we'll tell git we want to amend the previous one:

```
git add [FILENAME]
git commit --amend
```

We all know `git commit`, but the `--amend` flag is our friend here. This tells git that we want to edit the previous commit, rather than creating a new one. We can continue to make changes to the last commit in this way, right up until we're ready to either push our work, or start on a new commit.

Once you've amended the commit, you're done! The secret's gone, you can carry on as you were.

The secret is beyond the last commit

It's complicated

If you know you committed a secret, but have since committed other changes, things get trickier quickly. In anything but the simplest cases, we are going to want a more powerful tool to help us do a deep clean of the repository We're going to use `git-filter-repo`, a tool recommended by the git maintainers that will help us to rewrite history in a more user-friendly way than the native git tooling.

A technical aside to those familiar with the concept of rebasing. (If you don't know what that means, feel free to skip this paragraph.) All of the cases covered below can of course be managed using native git tools, particularly by rebasing. Sometimes rebasing is relatively painless, of course, but in the kinds of scenarios we're presenting here, rebasing is going to be a tedious and deeply error-prone process. This is why I am favoring purpose-built tools like git-filter-repo over rebasing. It is far better to avoid opening the possibility for making mistakes. From my own personal experience, recovering from a botched rebase is extremely time consuming, and often nearly impossible. Better to use the right tool for the job.

First, install

```
git-filter-repo
```

Next, let's assess the situation to determine which technique is right for your situation. Sometimes secrets are files, and sometimes secrets are lines of code. For example, if you accidentally committed an SSH key or TLS certificate file, these are contained in specialized files that you'll need to excise. On the other hand, maybe you have a single line of code containing an API key that's part of a larger source file. In that case, you want to modify one or more lines of a file without deleting it.

Git-filter-repo can handle both cases, but requires different syntax for each case.

Excise an entire file

To tell `git-filter-repo` to excise a file from the git history, we need only a single command:

```
git filter-repo --use-base-name --path [FILENAME] --invert-paths
```

The `--use-base-name` option tells `git-filter-repo` that we are specifying a filename, and not a full path to a file. You can leave off this option if you would rather specify the full path explicitly.

Normally, `git-filter-repo` works by ignoring the filenames specified (they are, as the name suggests, filtered out). But we want the inverse behavior, we want `git-filter-repo` to ignore everything except the specified file. So we must pass `--invert-paths` to tell it this. If you leave off the `--invert-paths` , you'll excise everything except the specified file, which is the exact opposite of what we want, and would likely be disastrous. Please don't do that.

Edit a file without removing it

If you only need to edit one or more lines in a file without deleting the file, `git-filter-repo` takes a sequence of search-and-replace commands (optionally using regular expressions).

But perhaps they need to be modified to prevent a runtime crash.
Next, create a file containing the search-and-replace commands, called `replacements.txt`. Make sure it's in a folder outside of your repo, for example, the parent folder.

The format of this file is one search-and-replace command per line, using the format:

```
ORIGINAL==>REPLACEMENT
```

For example, suppose that you've hard-coded an API token into your code, like this:

```
AUTH_TOKEN='123abc'
```

Now suppose that you've decided that it's better to load the API token from an environment variable, as such:

```
AUTH_TOKEN=ENV['AUTH_TOKEN']
```

We can tell `git-filter-repo` to search for the hard-coded token, and replace with the environment variable by adding this line to `replacements.txt`:

```
'123abc'==>ENV['AUTH_TOKEN']
```

If you have multiple secrets you need to excise, you can have more than one rule like this in `replacements.txt`.

Finally, assuming you placed `replacements.txt` in the parent directory, we invoke `git-filter-repo` with our search-and-replace commands like this:

```
git filter-repo --replace-text ../replacements.txt
```

Sometimes you might get an error saying you're not working from a clean clone. That's OK. Git-filter-repo is making irreversible changes to your local repository, and it wants to be certain that you have a backup before it does that. Of course, we do have a remote repository, and we're working from a local clone. And of course we are very interested in making irreversible edits to our commit history—we have a secret to purge! So there's no need for `git-filter-repo` to worry. We can reassure it that we are OK with making irreversible changes by adding the `--force` flag:

```
git filter-repo --replace-text ../replacements.txt --force
```

And now you have a clean git history! You'll want to validate your work by compiling your software or running your test suite. Then once you're satisfied that nothing is broken, move on to the next step to propagate the new history to your remote repository and the rest of the team.

Do you need to coordinate with your team?

No

If you only just added the secret, and haven't pushed any of your work yet, you're done. Just keep working like you had been, and no one will ever know. Don't forget if you need to restore uncommitted work by popping it from the stash with `git stash pop` .

Otherwise, we'll need to overwrite what's on your remote git repository (such as GitHub), as it still contains tainted history. We can't simply push, however: The remote repository will refuse to accept our push because we've re-written history. So we'll need to force push instead. Moreover, if our re-writes to history affect multiple branches or tags, we'll need to push them all up. We can accomplish all of this like so:

YES

If you work as part of a team, now comes the hard part. Everyone you identified as affected at the beginning of this process still has the old history. They need to synchronize against the revised history you just force-pushed. This is where errors can happen, and more importantly, where frustration can occur.

Ideally everyone pushed their work up before you edited the history. In that case, everyone can simply make a clean clone of the repo and pick up where they left off.

But if someone failed to push their work up before you re-wrote history, they're going to find they have a number of conflicts that need to be resolved when they pull. Instead, they need to fetch the new history from the remote repository, and rebase their hard work on the re-written history. To do this:

```
git fetch
```

```
git rebase -i origin/[branchname]
```

If you aren't familiar with `git fetch` , this command tells git to download new data from the remote repository, but unlike `git pull` , it doesn't attempt to merge new commits into your current working branch. So the fetch here is requesting all the newly re-written history.

I know I said I wasn't going to talk about rebasing, but this is the one instance where it's necessary. Once all the new history is pulled down, the developer will need to re-apply all their hard work on top of the re-written history. This is done by rebasing. Unfortunately, this can be a tedious and error-prone process, so care must be taken. For this reason, rebasing is out of scope for this tutorial. Instead, we recommend that you look at this tutorial for more information.