Lab 04 - Advanced Git

Objectives

- Setting up **gpg** for commit signing
- Applying a free-software license to your code
- Learning to use branches and to rebase commits

Contents

Tasks

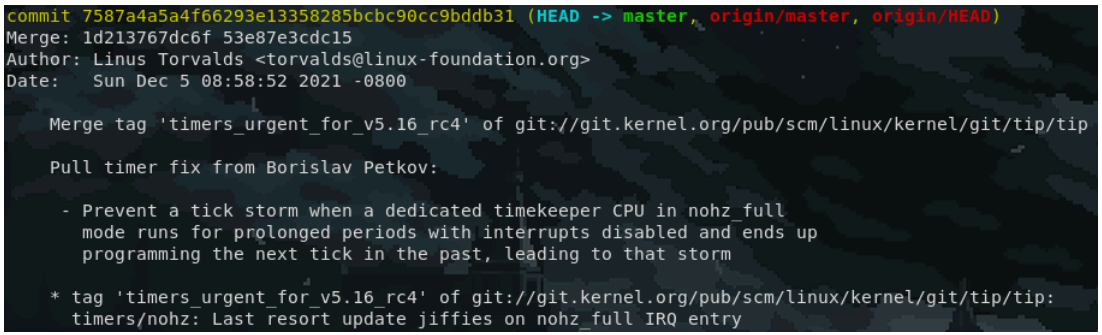
Introduction

By now, everyone should have gone over the 8th USO lab [https://ocw.cs.pub.ro/courses/uso/laboratoare/laborator-08]. So, you probably have a basic notion of how to use **git**. Our goal is to expand on what you already know from USO and discuss some practices that will hopefully be useful to you when deciding to contribute to open-source projects. In doing so, we'll inevitably stumble upon different subjects such as encryption, digital signatures and key management. Although related to the tasks at hand, these concepts are not relevant enough to spend a significant portion on this lab explaining them in detail. If you want to know more about one thing or another, just ask the assistant.

Tasks

01. [25p] Commit signatures

When looking at the `git log` of a repository, you would normally see a sequence of entries such as this:



Each commit is identified via a 40-character long **hexstring**. A hexstring is the hexadecimal representation of a sequence of bytes, where every nibble (i.e.: 4 bits) is represented by a character ranging from *'0'* (= 0b0000) to *'F'* (= 0b1111). But how is this identifier calculated?

Well… this commit identifier is called a **hash value** or a **digest**, and is the output of a hash function [https://en.wikipedia.org/wiki/Hash_function]. A hash function takes an arbitrary amount of data and outputs a fixed-size bit array that is representative of the input. Normally, one would be weary of collisions: if the function's domain is virtually infinite and the co-domain is not only finite but rather small in comparison, wouldn't it be possible to create two commits with the same digest? Possible – yes, likely – no. Git uses **SHA1**, a cryptographic hash function [https://en.wikipedia.org/wiki/Cryptographic_hash_function]. What makes a cryptographic hash function so special is that it provides certain guarantees. For example, it should be impossible to calculate potential messages from a digest (meaning that the function is non-invertible). Moreover, any change in the input – no matter how small, should change the hash value so extensively that the new value and the old should appear uncorrelated. Consequently, being able to *craft* a commit such that it's not only comprehensible, but also creates a certain desired digest is so unlikely that it occurring naturally should not pose any risk.

There is, however, a more significant risk here. What guarantee do you have that the author of the commit above is actually Linus himself? Using something like git-blame-someone-else [https://github.com/jayphelps/git-blame-someone-else], you could overwrite commits and change their author only to then `git push --force` and replace the remote history (don't do this on the master branch if you're not working alone). The answer to this problem is **commit signing**. Cryptographic algorithms can be loosely categorized an **symmetric** and **asymmetric**. Symmetric algorithms like AES [https://en.wikipedia.org/wiki/Advanced_Encryption_Standard] use the same key for both encryption *and* decryption. Asymmetric algorithms like RSA [https://en.wikipedia.org/wiki/RSA_(cryptosystem)], on the other hand, utilize two keys. One for encryption and one for decryption. Usually, the one used for encryption is called the private key and the one used for decryption, the public key. The private key is your identity, so you don't share it with anyone. The public key you configure on remote servers to give them the ability to verify your identity (e.g.: configuring SSH keys on fep.grid.pub.ro). As a rule, you use asymmetric cryptography in cases where you need to prove your identity to a remote host, establish secure communication channels over an untrusted network, etc. The reason for this is that asymmetric algorithms are orders of magnitude slower than their symmetric counterparts. While encrypting the output of a **SHA256** function (32 bytes) with a 4096-bit RSA key takes about 5ms on regular CPUs, it takes almost a full minute on an Arduino Mega (with a MCU running at 16MHz). AES-256 on the other hand, encrypts the same amount of data in less than 5 microseconds. The downside is that you need to share your key with other systems for them to extract the plaintext message.

In the following tasks we will introduce **GNU Privacy Guard (gpg)** an open source encryption and signing tool. **git** can use **gpg** to sign your commits as you create them. For it to work, you will also have to upload your public key to github [https://github.com/settings/keys].

[15p] Task A - GNU Privacy Guard

As we mentioned before, **gpg** is an encryption and signing tool. Additionally, it also handles key management. When you first create a keypair, it will also construct you a **key ring**. This key ring can store both private keys (i.e.: your keys – you can have more than one) and multiple public keys. These public keys can belong solely to you, or to persons that you trust.

Let's say that you have *Person A*'s public key and *Person B* has your public key. If you want to introduce *Person A* to *Person B*, you can give his public key (that you have on your key ring) to *Person B*. If you're doing this over a secure channel (e.g.: you write it on a piece of paper and then hand it to *Person B*) all's well and good. However, if you send it over an untrusted medium (e.g.: the Internet), you should sign *Person A*'s public key with your private key. As a result, *Person B* can validate the integrity of *Person A*'s key with your public key.

This example with *Persons A and B* is characteristic of the Web of Trust [https://en.wikipedia.org/wiki/Web_of_trust] model. In it, every participant is responsible not only for his own key, but for those that he decides to add to his key ring – and thus, implicitly trust. As you can imagine, validating each new person (or server) that you communicate with can become taxing. Consequently, it's no surprise that the most prevalent model today is the Public Key Infrastructure [https://en.wikipedia.org/wiki/Public_key_infrastructure]. Further discussing the differences between these two models falls beyond the scope of this lab. However, this is a topic that you should research sooner rather than later (so don't wait for it to be mentioned during lectures).

For now, let's get you started by generating a brand new 4096-bit RSA keypair. Make sure to introduce your real name and email. At some point, you will be prompted for a password. That password is to secure your key ring. Its only purpose is to prevent attackers that have gained access to your computer from using your keys. Otherwise, the password will not factor in any of the cryptographic operations that **gpg** performs.

```
# generate key
$ gpg --full-generate-key

# list keys
$ gpg --list-public-keys
$ gpg --list-secret-keys
```

Now, let's upload your public key to https://github.com/settings/keys [https://github.com/settings/keys]. Click on `New GPG key` and copy paste the output of the following command:

```
# export public key
$ gpg --armor --export ${YOUR_MAIL_GOES_HERE}
```

Notice the `--armor` flag. Without it, the command above would write your key in a binary format to *stdout*. With it, **ASCII armor** is applied to the output. ASCII armor is a technique that encodes your binary data using algorithms such as base64 [https://en.wikipedia.org/wiki/Base64] in order to make it ASCII-printable. Because only a subset of 8-bit values are printable, a number of bits smaller than 8 (e.g.: for b64 encoding that number is 6) are expanded to a full ASCII printable byte. Naturally, this leads to output inflation but at the very least you can copy paste data or send it via email without attachments.

From now on, you can add the `-s, --signoff` flag to `git commit` and it will automatically add a *"Signed by …"* message at the end of your commit message and sign it with your default private key. Alternatively, if you don't want that trailing message, you can use `-S, --gpg-sign[=<keyid>]`. If you want to configure **git** to automatically sign all commits with a certain key, you can do so thusly:

```
# get your private key fingerprint (long upper case hexstring)
$ gpg --list-secret-keys

# configure your default signing key (git user.{name,email} must match those in key)
$ git config --global user.signingkey ${PRIVATE_KEY_FINGERPRINT}
```

```
# enable autosigning
$ git config --global commit.gpgsign true
```

[10p] Task B - Your first signed commit

Create a public repository on GitHub and add your music bot from lab 3 as a first commit. Make sure you use `git commit -s`, then push your changes to remote. Go to the *"github.com/…/commits"* page of your repo and check that your commit has a green *"Verified"* tag next to it.

Troubleshooting

If `git commit -s` fails, there are usually two reasons. Prepend `GIT_TRACE=1` to the command and run it again. Look for the last command executed in the background by **git** before the error occurred (most likely an invocation of **gpg**) and run it separately. This will give you a more verbose error that will let you determine the problem. Here are the usual suspects:

- The `user.name` and `user.email` values that you previously configured with `git config [--global]` differ from what you specified when creating the **gpg** public keypair. Remember that `user.name` is not the same as your account username on GitHub. It is the name under which you publish your commits and can be whatever you want it to be, but be consistent!
- **gpg** has no idea how to ask you for the passphrase needed to decrypt your keyring. This might be an issue when invoking it indirectly from **git**, usually from wsl [https://docs.microsoft.com/en-us/windows/wsl/install]. Add `export GPG_TTY=$(tty)` to your *.zshrc* or *.bashrc* and source it. This environment variable will let **gpg** know that it should use an ncurses [https://en.wikipedia.org/wiki/Ncurses#:~:text=ncurses%20(new%20curses)%20is%20a,runs%20under%20a%20terminal%20emulator.]-based terminal prompt. Note that after you input your passphrase once, it will be temporarily cached by a daemon named **gpg-agent**. As a last resort, try running `gpg --sign ${SOME_FILE}` just so `git commit -s` won't ask you for it afterwards. Again, this is only a workaround, not a solution. Ask your assistant for help!

02. [25p] Choosing a license

If you ever decide to publish code that you've written, note that it will automatically fall under the protection of copyright law. This means that distributing copies of your code, or using it as a basis for something that may be construed as derivative work is prohibited. As a result, people will generally stay clear of your project since they don't know what your intentions are.

In the 1980s, Richard Stallman [https://www.youtube.com/watch?v=jUibaPTXSHk] pioneered concepts known as free software and copyleft. Free software is software distributed with a guarantee that the end user can modify and adapt it for whatever purpose, profit included. In order for this to happen, the user must have ultimate control over the software in question, which implies access to the source code. So, for a piece of software to become *"free software"* it must include a public license such as the GNU General Public License [https://www.gnu.org/licenses/gpl-3.0.html], MIT license [https://opensource.org/licenses/MIT], etc. These licenses waive part of the author's rights and and grants them to the recipient of the software. Almost all free-software licenses contain a copyleft provision. This provision states that when modified versions of the free software are distributed, it must provide the same guarantees as the original, under the same license (or a more permissive one).

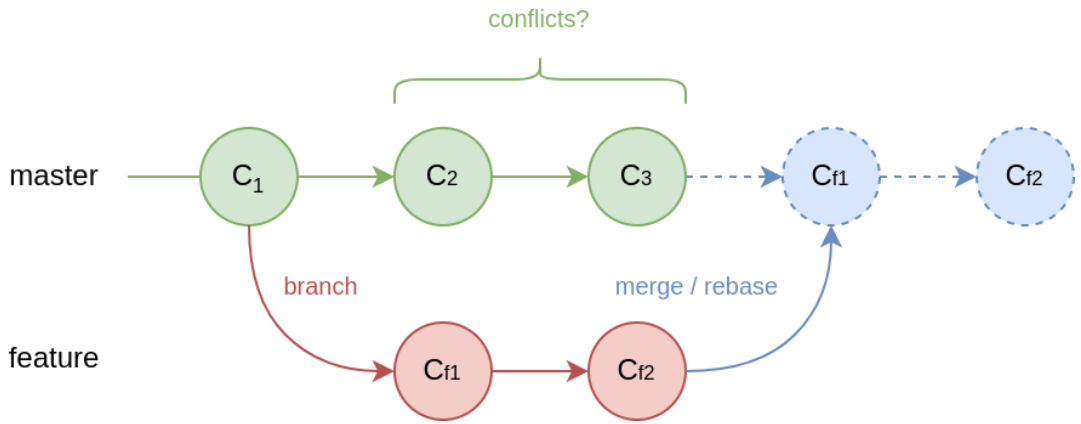
In this exercise you will *manually* add a GPL license to your bot. If you want to learn more about different kinds of licenses (or licenses in general), listen to this episode of the Destination Linux podcast. If you want to go straight to the discussion on each individual license, skip ahead 10m relative to the timestamp in the video.

[25p] Task A - Adding a GNU license

Following these indications [https://www.gnu.org/licenses/gpl-howto.en.html] on how to use GNU licenses, add GPLv3 to your project. Create a single commit with all the changes (i.e.: COPYING file, copyright notice, etc.) and push it. You don't need a copyright disclaimer from the school, so don't worry about that part.

03. [50p] Adding & changing features

When you want to add a new feature to your project, you should first develop it in a branch [https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-branches]. A branch is a named copy of the deltas that comprise your codebase up to a certain point. By adding commits to this copy, you won't interfere with other people trying to do their own thing. Note that a branch can be created from any other branch, including *master*.



Eventually, you will want to merge your commits with the original branch. This can be done in two ways: `git merge` or `git rebase`. Here is a discussion [https://www.atlassian.com/git/tutorials/merging-vs-rebasing] on which is better. You should probably read it at some point. In this lab we will be focusing on `git rebase` since it is more interactive and provides many functionalities that you will need when trying to get your changes accepted

by the maintainer / reviewer.

[30p] Task A - Add token flag

The feature that you'll want to add to your project is a command line argument parser that will accept an optional `-t, --token [TOKEN]`. We suggest that you use argparse [https://docs.python.org/3/howto/argparse.html]. In absence of this token, you will fall back to fetching it from the environment variable.

```
# first, create a new branch from HEAD
$ git branch feature

# next, switch to the feature branch
$ git checkout feature

# check that the branch you are on is actually feature and not master
$ git branch
  * feature
    master

# edit and test your script
# argparse should add a default '--help' option

# commit changes and push them to the remote feature branch
# first push means that the branch needs to be created (follow the command's hints)
$ git add ${BOT_SCRIPT}
$ git commit -s
$ git push
```

At this point, you have created a separate *feature* branch, added a (hopefully) working CLI argument parser, and pushed the newly created branch to your *remote*. When working with other people, now would be a good time to create a Pull Request [https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/creating-a-pull-request] (PR). This is a request to the maintainer of the project to pull your *feature* branch, check that everything is working, and give feedback if changes need be made. If changes are indeed requested, all you have to do is address them in a new commit which you'll push into your *feature* branch. The PR will be updated automatically. Once the reviewer gives his ok, your changes will be applied to the *master* branch.

Since this is your repository and you have to deal with integrating the changes, let's use `git rebase` to do just that.

```
# switch back to the master branch
$ git checkout master

# apply the extra commits from feature onto master
$ git rebase feature
Successfully rebased and updated refs/heads/master.

# remember to push the newly integrated changes to remote
$ git push
```

"Wait. That's it?" Well… yeah. Luckily, you did not have any conflicts with *master*. If you did, `git rebase` would have told you exactly where those conflicts were located. Moreover, it would have modified your files to look something like this:

```
<<<<<<< HEAD
Changes made to master since branch.
=======
Changes made to feature since branch.
>>>>>>> feature
```

In order to resolve the conflicts, you would have to remove the lines with *"<<<"*, *"==="*, *">>>"* and rewrite the conflicting code so that it incorporates both your changes, and those already pushed to *master*. Finally, mark the conflicts as resolved by re-adding the files, and continue your rebase.

```
# re-add files with solved conflicts
$ git add ${CONFLICTING_FILES}

# continue the rebase process
$ git rebase --continue
```

```
# alternatively, you can just give up and go back to how things were (no harm done)
$ git rebase --abort
```

This part is now optional, but it would be nice to clean up and delete the *feature* branch both locally and remotely. All changes that *feature* held are now part of *master*, so what is it good for anymore?

```
# delete feature branch on remote (origin)
$ git push -d origin feature

# delete feature branch locally
$ git branch -d feature
```

[20p] Task B - Edit older commits

In the beginning we said that `git rebase` is interactive and fun. But we never had the chance to show it. Remember in the previous exercise when we added the COPYING file and the copyright notice to the *Python* script? Let's say that the reviewer changed his mind about this and now wants us to create two separate commits. One for the script and one for the copy of GPLv3. How would we go about solving this problem?

Click GIF to maximize.

Why, using `git rebase`, of course!

```
# take a look at the commits we have so far
#    #1: adding the bot
#    #2: adding the GPL license
#    #3: adding the argument parser
$ git log

# launch git rebase in interactive mode (-i)
# and tell it we want to revisit the last 2 commits relative to our head
$ git rebase -i HEAD~2
```

After running `git rebase -i`, it should have opened your default CLI file editor (same as with `git commit`). Notice that you have two lines that look something like this, followed by multiple lines describing commands.

```
pick 4864b9d Added GNU General Public License.
pick 37d816c Added cli argument parser for token.
```

Once we save this file, **git** will parse it's non-comment contents line by line and execute the commands on the given commits, in the order that they were specified. The **pick** command just selects a certain commit. By swapping lines, you will tell git to **pick** commits in a different order, thus reordering them on your current branch. Deleting a line will effectively delete the changes made by that commit in the current repository. What we're interested in, however, is the **edit** command. This command tells **git** to stop the rebasing process at that specific commit and let you make changes to it before proceeding.

```
# HEAD is now on commit 4864b9d which we marked for edit

# revert the changes made with this commit ==> files no longer added in staging area
$ git reset HEAD~1

# check the status of the files; see how COPYING and the Python script are now untracked
$ git status

# add the files one at a time; and commit them separately
$ git add COPYING
$ git commit -s

$ git add ${BOT_SCRIPT}
$ git commit -s

# from one commit, we now created two; continue the rebasing process
$ git rebase --continue

# check to see that two new commits were indeed created
$ git log

# push this changes to remote, thus rewriting history
$ git push --force
```

Once again, force pushing a different commit history onto the *master* branch is a bad idea if working with other people. But doing it onto your own branch is not only fine, but sometimes necessary in order to address the reviewer's requests.

04. [10p] Feedback

Please take a minute to fill in the feedback form [https://forms.office.com/Pages/ResponsePage.aspx?id=usiMLdqNNEOeXPrCCS6brJoxNMaLqNZHpd8YaA7IhDNUNVVLQ0lQV0tJRzBaRjhOQzdNOVhYWklBVC4u] for this lab.