Write a fuzzer

The goal of this work is to write a *fuzzer* for a simple implementation of a *tar* extractor. [1]
This project must be done by **groups of max. 2 students**. Submitting alone is also accepted but should be avoided as much as possible.

The *tar* format

`tar` is a file format for file archives. It concatenates files and custom headers in a whole archive.
The description of those headers is available at
https://www.gnu.org/software/tar/manual/html_node/Standard.html.
For this project, we use the **POSIX 1003.1-1990** format.
To create an archive corresponding to this format you can use

```
tar --posix --pax-option delete=".*" --pax-option delete="*time*" --no-xattrs --no-acl --no-selinux -c fichier1 fichier2 ... > archive.tar
```

Your fuzzer **may not** use this command.

To visualize the hexadecimal content of the archive, use the following command:

```
hexdump -C archive.tar
```

What is a fuzzer?

The *tar* extractor works correctly for input files matching the aforementioned specification.
However, it crashes sometimes if the input file is not correctly formatted. In that case, it writes

```
*** The program has crashed ***
```

on stdout.
This is of course very dangerous. Imagine what could happen if such a vulnerable tool is run on a web server, for example on INGInious that would allow students to upload their code as tar archive during an exam.
Security experts sometimes use *fuzzing* tools to find vulnerabilities in programs. A fuzzer is a tool that generates input data with the goal of crashing the tested program. When such input data is found, it is saved so it can be analyzed later by security experts.
There are different types of fuzzers (https://medium.com/@elniak/mastering-fuzzing-a-comprehensive-tutorial-ba9431c8ff0f?sk=91edf37ec087af2aaba00178a5fbaf46):

1. In the simplest form, a fuzzer generates purely random input files. Such a fuzzer is easy to write but quite inefficient: Most input files would probably not be accepted by the tested program because they have the wrong format.
2. *Mutation-based* fuzzers take a valid input file and modify it slightly, for example, by adding additional bytes at random places.
3. Generation-based fuzzers generate valid input files based on the knowledge of the input format. To find vulnerabilities in the tested program, they often test extreme cases (e.g. very large numbers in an input field, etc.).

Your job

Your job is to write a generation-based fuzzer for a *tar* extractor. The fuzzer should automatically generate input files (not mutate a given one) and check whether the extractor crashes. Input files that successfully crash the extractor are saved.

1 The *tar* extractor is under MIT License (available at https://opensource.org/licenses/MIT) Copyright 2022 Tom Rousseaux, Ramin Sadre

We give you a toy-example extractor that can be downloaded here for the x86_64 architecture, or here for the Apple Silicon chip. Depending on your architecture, the extractor is executed by one of the two following commands:

```
./extractor_x86_64 archive.tar
./extractor_apple archive.tar
```

where archive.tar is the tar file and sources are the files you want to extract.
NB: you might need to execute the command `chmod +x extractor_name` to be able to execute the program.

We are aware of at least six different ways to crash this toy-example extractor, i.e. when the program writes the crash message. "Different ways" means that they should not just be variations of the same vulnerability. For example, if you have discovered that the tool crashes for all no-ascii name field, input files with `name`=\x90\x00, `name`=\x90\x90\x00 etc., only count as one way.

Be aware that your fuzzer **will be tested on another extractor** than the one we give you. It means that if you discover that the program crashes for `typeflag`=\x90, you cannot hardcode this value in your fuzzer. In the examination version, maybe the program will crash for `typeflag`=\x91, maybe it will not crash for any value of `typeflag`.

Your fuzzer must be smart: you cannot try every value for every field. Trying all the values for the field name would imply formatting $(2^8)100$ archives, which is not sustainable. For example, you can soundly assume that if the name field accepts every non-ascii value at every position of the string, combining every string of some non-ascii characters is useless. This will allow your fuzzer to run in a reasonable time.

Your fuzzer must work with archives: you cannot just try different values for different fields in the header. You must deal with headers with and without data, with archives containing multiple files, etc.

Deliverable

Submission will be done on INGInious: https://inginious.info.ucl.ac.be/course/LINFO2347/project
You must register to a group on INGinious with your partner to submit.
You must upload a zip file before Friday, March 7th, 14:00, containing exactly:

- The **commented** source code of your fuzzer.
- A Makefile which compiles your project to an executable named *fuzzer*. The generated executable takes one argument: the path to the tar extractor.

You must implement your fuzzer in C, version 99 or later. The source code must be compilable with gcc, on a 64-bit x86-64 Linux system without any additional dependencies other than the standard libraries. If you don't own a computer with such architecture, you can use the machines in the student rooms. To test whether your fuzzer works correctly, we will copy the executable into a directory together with an extractor and start it from there. Your fuzzer may not rely on any other file. The input files must be generated in the same directory. We will assume that all generated files with a name starting with "success_" contain successful input files (i.e. files that crash the extractor).

Implementation hints

You do not have to perform fuzzing on the fields *devmajor*, *devminor*, *prefix* or *padding*.
The file *help.c* contains:

- The header structure you are supposed to use.
- An example of code that launches a program given as argument, parses its output and checks whether it matches the crash message "`*** The program has crashed ***`".
- A function that computes the checksum of a header and writes it in the *chksum* field.