Write a fuzzer

The goal of this work is to write a *fuzzer* for a simple implementation of a *tar* extractor. [1]

The *tar* format

Tar is a file format for archive. It concatenates files and custom headers in a whole archive.
The description of those header are 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 **should not** use this command.

To visualize the archive you can use

```
filename -C archive.tar
```

What is a fuzzer?

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

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

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 to crash the tested program. When such input data is found it is saved so it can be analyzed later by security expert.
There are different types of fuzzers (https://www.f-secure.com/en/consulting/our-thinking/15-minute-guide-to-fuzzing):

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 the *tar* extractor. The fuzzer should automatically generate input files and check whether the converter crashes. Input files that successfully crash the converter are kept by the fuzzer.
The extractor is called by

```
./extractor archive.tar [sources]
```

where *achive.tar* is the tar file and *sources* are the files you want to extract. If *sources* is empty, all the files will be extracted.
We are aware of at least six different ways to crash (i.e. the program writes the crash message) the converter.

---

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

“Different ways” means that they should **not be just 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 crash 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 has to 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 accept 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 has to work with archives: **you cannot just try different values for different fields** in the header. You have to deal with header with and without data, to have multiple files in an archive etc.

Deliverable

You have to upload a zip file to INGInious containing exactly:

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

The INGInious task will come shortly. It checks that your archive matches the instructions but **it do not grade your project.** Grades will come after the deadline.

You can implement your fuzzer in version 99 of C or later and it should be compiled with gcc. The source code must be compilable/runnable on a 64-bit x86-64 Linux system without any additional dependencies other than the standard libraries. To test whether your fuzzer works correctly, we will copy it into a directory together with an extractor and start it from there. 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 converter).

Implementation hints

You do not have to do 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, parse its output and check whether or not it matches *** The program has crashed ***.
- A function that computes the checksum of a header and write it on the *chksum* field.