Write a fuzzer

The goal of this work is to write a *fuzzer* for a simple program called *Converter*.

The program *Converter*

Converter is a small tool written in C that reads a picture with indexed colors and converts it into a picture with non-indexed colors. The tool is started like this:

```
./converter  inputfilename  outputfilename
```

Converter only supports the ABCD image file format (invented just for you!). An ABCD image file always starts with the two bytes 0xAB 0xCD followed by two bytes indicating the version number as a 16-bit value. The current version is 100, although some older versions are also accepted by the tool. Note that the ABCD file format is a *binary* format where numbers are stored in *Little-Endian*.

The version number is followed by one or more header fields. Every header field starts with one byte identifying the header field, followed by the data for that header field. Supported header fields are:

Field ID	Data
1	Author name. A sequence of 8-bit characters. Terminated with a 0-byte.
12	Comment. A sequence of 8-bit characters. Terminated with a 0-byte.
2	Width and height (in pixels) of the picture. Two 32-bit values.
10	32-bit value. The number of colors in the color table (see the next field). In the current version, we only support color tables with up to 256 colors.
11	The color table. A sequence of 32-bit color values.
0	No data. Indicates the end of the list header fields.

The header with ID 0 must be always the last header field. The rest of the file contains the actual image data. It is a sequence of width*height unsigned 8-bit values, representing the colors of the pixels. Each value is interpreted as an index in the color table.

Here is an example of a 2x2 image (i.e. 4 pixels) with 2 colors 0x00000000 (black) and 0x00FFFFFF (white). For better readability, the bytes indicating the header field ID are underligned.

```
AB CD                        // the magic number in Little Endian
64 00                        // version = 100
02 02 00 00 00 02 00 00 00   // width = 2, height = 2
0C 48 65 6C 6C 6F 00         // comment = “Hello”
01 52 61 6D 69 6E 00         // author = “Ramin”
0A 02 00 00 00               // the size of the color table. The color table contains two colors.
0B 00 00 00 00 FF FF FF 00   // the colortable with the two colors
00                           // end of header
00 01                        // the 4 pixels of the image:     black   white
01 00                        //                                white   black
```

The tool reads the input image file and produces an output image file where the pixels with the color indexes are replaced by pixels with the full 32-bit color values from the color table. The output version is always 100 and the output file does not contain a color table. On Linux, you can look inside the files with

```
hexdump -C filename
```

What is a fuzzer?

The *Converter* tool works as described above for correct input files. 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 a social network website, that allows users to upload their own image files.

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 is to write a generation-based fuzzer for the *Converter*. 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.

We are aware of at least six different ways to crash (i.e. the program writes the crash message) the converter. Your fuzzer should be able to find example input files for five of them.

Deliverable

You have to upload a zip file to Moodle containing exactly seven files:

- The commented source code of your fuzzer in a single file.
- Five "bad" input files that show five different ways to crash the converter tool. "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 width>30, input files with width=31, width=32 etc., only count as *one* way.
- A readme file explaining what the five bad input files do. The explanations must be short, for example "Input file 2 crashes the tool by using width=40". Don't write a full report.

You can implement your fuzzer in Java (version 9 or later), C (version 99 or later, compiled with gcc), or python (version 3 or later). The source code must be a single file called fuzzer.java|c|py and be compilable/runnable on a 64-bit x86-64 Linux system with default compiler switches and 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 the converter tool 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).

Do not hardcode crash values directly into your fuzzer. That means, if you have heard from friends for example that the converter crashes for width=31 on their computer, do not write a fuzzer that only generates files with width=31. Instead, test different values. We will also test your fuzzer with variations of the converter tool that crash for example for width>40 instead of width>30.

Your solution will be evaluated according to the following criteria:

- Quality and readability of your source code and the readme file.
- Correctness of the solution.

Implementation hints

To start an external process (i.e. the converter) and read its output from your tool:

- Java: Use the Process object returned by Runtime.getRuntime().exec(). See also https://stackoverflow.com/questions/808276/how-to-add-a-timeout-value-when-using-javas-runtime-exec if you want to limit the execution time.
- Python: https://docs.python.org/3/library/subprocess.html#subprocess.Popen.communicate