Group Work: Fuzzer

In one of the lectures, we briefly mentioned "Fuzzers", i.e. tools that try to find vulnerabilities in programs by testing them with random input. The goal of this group 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. We explain in the following what that means.

Converter only supports one file format, the ABCD image file format (invented at UCLouvain just for you). An image file in this format looks like the following:

0xAB 0xCD	A so-called magic number. All ABCD files start with these two bytes
version	16-bit value. Version number of the file format. Current version is 100, although some older versions are also accepted.
authorname	A sequence of 8-bit characters. Terminated with a 0-byte.
width	32-bit value. The width of the picture (in pixels). A negative value indicates a compressed file but this feature is not supported by the current version of the tool.
height	32-bit value. The height of the picture (in pixels)
numcolors	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.
colortable	An array of 32-bit color values
pixels	A sequence of unsigned 8-bit values, representing the colors of the pixels. Each value is interpreted as an index in the color table.

The ABCD file format is a *binary* format where numbers are stored in *Little-Endian*. That means that a 2x2 image with 2 colors 0x00000000 (black) and 0x00FFFFFF (white) and author name "Ram" contains the following 30 bytes:

```
AB CD                  // the magic number in Little Endian
64 00                  // version = 100
52 61 6D 00            // author = Ram
02 00 00 00            // width = 2
02 00 00 00            // height = 2
02 00 00 00            // the size of the color table. The color table contains two colors.
00 00 00 00            // the first color
FF FF  FF 00           // the second color
00 01                  // the 4 pixels of the image:        black   white
01 00                  //                                   white   black
```

The tool is started like this:

```
./converter  inputfilename  outputfilename
```

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.

```
CD AB                          // the magic number in Little Endian
64 00                          // version = 100
52 61 6D 00                    // author = Ram
02 00 00 00                    // width = 2
02 00 00 00                    // height = 2
00 00 00 00                    // no color table (color table size is 0).
00 00 00 00 FF FF  FF 00       // the 4 pixels of the image:        black   white
```

```
FF FF  FF 00 00 00 00 00    //                                                  white   black
```

The folder on Moodle contains an example input file “testinput.img” with 16x16 pixels and 4 colors and also the corresponding output file “testoutput.img”. On Linux, you can look inside these files with

```
        hexdump -C filename
```

The folder also contains the *Converter* tool, compiled for Linux on Intel CPUs in a dynamically linked version (“converter”) and a statically linked version (“converter_static”). Depending on your Linux distribution, the dynamically linked version might refuse to start.

Your job: The fuzzer

The *Converter* tool works as described for correct input files. However, it crashes sometimes if the input file is not correctly formatted[1]. 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 (sometimes randomly) 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 the security expert.

Your job is to write such a fuzzer for the *Converter*. The fuzzer should automatically generate input files and check whether the converter crashes. If the converter crashes, the input file should be kept.

In fact, you have to write *two* fuzzers:

Fuzzer 1: Mutation-based fuzzer

Real programs (like OpenOffice) have so complex input files that it is difficult to write a fuzzer that tests all possible byte sequences. For this reason, many fuzzing tools generate random input data, hoping to find a combination that crashes the program. However, it is quite difficult to write fuzzers that produce useful input files. For example, the *Converter* tool only accepts input files that start with 0xABCD. A completely random fuzzer would generate a lot of useless input files.

To avoid this problem, a *mutation-based* fuzzer takes a correct input file[2] and slightly modifies it randomly, tests it, modifies it randomly if it did not crash the program, tests it, etc. We call such a sequence of attempts a *test run*.

You should write a fuzzer that accepts three parameters on the command line, like this:

```
        ./myfuzzer mycorrectinputfile.img 200 100 0.01
```

where

- “mycorrectinputfile.img” is the name of the correct input file,
- 200 is the number of test runs to make,
- 100 is the maximum number of modifications to make in one test run before giving up and starting a new test run with the correct input file,
- 0.01 means that the fuzzer randomly changes 1% of the bytes in the input file in order to create the next input file for the current test run. Input files that crash the Converter tool should be kept.

Fuzzer 2: Generation-based fuzzer

The *Converter* tool has a very simple file format and you can probably already guess which fields you have to modify to make the tool crash, just by reading the description of the tool. In such situations,

[1] When the tool crashes it writes

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

on stderr.

[2] Such a file is called an *input seed*.

it might be better to use a fuzzer that knows how the input format looks like and that can therefore directly create files in the correct format. Such a *generation-based* fuzzer can be much faster than a mutation-based one.

Write a fuzzer that directly generates new ABCD input files without using mutation. Of course, your program should not try all possible values. Write the fuzzer such that it tests only interesting values. It's up to you to decide what the interesting values could be.

We are aware of at least five *different* ways to crash the converter. Your fuzzer should be able to find example input files for all of them in a *reasonable* amount of time.

Deliverable

You have to upload a zip file to Moodle containing:

- The commented (!) source code of your mutation-based fuzzer and your generation-based fuzzer. You can implement your fuzzer in Java, C, C++ or python.
- Five "bad" input files that show five different ways to crash the converter tool. "Different ways" means that there should not be just variations of the same vulnerability. For example, if you have discovered that the tool crashes for width=32 and width=33, this only counts as *one* way, not two.
- A readme file explaining (a) how to compile and use your fuzzers and (b) what the five bad input files contain. The explanations can be short, for example "Input file 2 crashes the tool by using width=32". Don't write a full report.

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.