IT Security 2024/2025
Exercise Sheet 7
– Fuzzing –

Christian Hartlage

Publication: 21.11.2024
Deadline: 27.11.2024 10:00

Lightning Survey ⚡

In the following five tasks, your knowledge about the theory and practice of fuzzing will be tested. The tasks partly build on each other, so it makes sense to do them in ascending order. Task 1 and Task 2 are of theoretical nature and can be solved without doing any actual instrumentation or compilation. Task 3, 4 and 5 are supposed to be done in a `x86_64` Linux build environment. If you don't have that handy, you can use the docker image we prepared. It's based on the official Debian base image and has been enhanced with recent versions of AFL++ and libFuzzer.

Find it at:	`https://hub.docker.com/r/hartlage/unibn-macs-itsec-fuzzing`
Pull it like this:	`docker pull docker.io/hartlage/unibn-macs-itsec-fuzzing`
Use it like this:	`docker run -it docker.io/hartlage/unibn-macs-itsec-fuzzing bash`

If you want to save artifacts that are being generated in the docker container (like executables or instrumented libraries) you will probably want to mount a directory from the host like this:
`docker run -v /home/hartlage/data:/data -it [...]`

Read more about docker volumes here: `https://docs.docker.com/storage/volumes/`

Exercise 1 (Do some research about fuzzers, 2 points)**.** Find out the names of five fuzzers that were not mentioned in the lecture. You can use resources that were provided in the lecture, of course. Write the name of the fuzzers in a file called `five_fuzzers.txt`. You should separate the fuzzer names with newlines.

Exercise 2 (libFuzzer and AFL++ Basics, 1+1 points)**.**
Part 1: Imagine you are given a CMake project that has no external dependencies, meaning no shared object or static library that has to be linked. You want to fuzz the project's library and somebody already prepared a working fuzztarget in a file called `fuzzer.c`.
Your tasks are:

(1) Instrument the CMake project for libFuzzer. *(Remember you need the right flags for library instrumentation as well as the right compiler)*

Write the command line to prepare the instrumentation in a file called `commandline_cmake_libfuzzer.txt`.

(2) Build the fuzzer that is defined in `fuzzer.c`. The executable fuzzer built by the compiler should be called `fuzzer`. *(Remember the flags for building a fuzzer are just slightly different than the flags for instrumenting the library)*

Write the command line to build the fuzzer in a file called `commandline_fuzztarget_libfuzzer.txt`.

Part 2: Imagine you are given a CMake project that has no external dependencies, meaning no shared object or static library that has to be linked. You want to fuzz the executable that is built by the project. The executable built by CMake is called `program` and receives its input via STDIN.
Your tasks are:

(1) Instrument the CMake project for libfuzzer. *(Remember you need the right compiler)*

Write the command line to prepare the instrumentation in a file called `commandline_cmake_afl.txt`.

(2) Fuzz `program` with AFL++.

You can assume AFL++ is installed globally, `afl-fuzz` is available through `$PATH` and `program` is in your current working directory.

Write the command line to fuzz `program` with `afl-fuzz` in a file called `commandline_fuzztarget_afl.txt`.

Exercise 3 (Prepare libzint with libFuzzer, 1+1 points).

(1) Download a copy of libzint from `https://github.com/dende/zint` and instrument it with libFuzzer. Find out which build tool is being used and apply the respective methods for instrumentation.

By default, building the project will yield a shared object containing the library's functions. Rename this file to `libzint_libfuzzer.so` and submit the file.

(2) Download a copy of libzint from `https://github.com/dende/zint` and write a fuzztarget for it.

Now that you have instrumented the library for libFuzzer, write a fuzztarget for it. Submit your fuzztarget as `fuzztarget_libfuzzer.c`.

Hint: libzints main encoder entry points seem to be in a file called `library.c` in the `backend` folder.

Exercise 4 (Instrument libzint for Fuzzing with AFL++, 2 points).
Download a copy of libzint from `https://github.com/dende/zint` and instrument it with AFL++. Find out which build tool is being used and apply the respective methods for instrumentation.

By default, building the project will yield a shared object containing the library's functions. Rename this file to `libzint_afl.so` and submit the file.

Exercise 5 (Find malicious inputs, 2 points).
The provided version of libzint (2.7.1) contains a bug in the EAN encoder. Since you just prepared the libzint for fuzzing with libFuzzer and AFL++ use the fuzzer of your choice to try to identify some malicious inputs that crash the library. Once you identified the bug, you can either continue to fuzz the library to generate 10 malicious inputs or if you understood what exactly causes the bug, you can of course also generate the inputs manually.
Write 10 malicious inputs in a file called `10_triggers.txt` and separate them by newlines, e.g.:

```
first input
this also crashes libzint
ohnoitcrashed again
128073918471
128073918471+012
[...]
```