CISPA HELMHOLTZ CENTER FOR INFORMATION SECURITY

Security Testing

WS 2021/2022

Prof. Dr. Andreas Zeller
Leon Bettscheider
Marius Smytzek

Exercise 11 (10 Points)

Due: 30. January 2022

> The lecture is based on The Fuzzing Book, an *interactive textbook that allows you to try out code right in your web browser*. The Fuzzing Book code is additionally available as a Python pip package. To work on the exercises, please install the package locally:
> `pip3 install fuzzingbook`

Submit your solutions as a Zip file on your status page in the CMS.

We will provide you a structure to submit your solutions where each task has a dedicated file. You can add new files and scripts if you want, but you may not delete any provided ones. You can verify whether your submission is valid by executing `verify.py`:

```
python3 verify.py
```

The output provides an overview if a required file, variable, or function is missing and if a function pattern was altered. If you do not follow this structure or change it, we cannot evaluate your submission. A non evaluable exercise will result in 0 points, so make sure to verify your work before submitting it. Note that the script does not reveal if your solutions are correct.

Exercise 11-1: Configuration fuzzing (8 Points)

In this exercise you will apply the approach to extract a grammar from an argument parser that was presented in the fuzzingbook chapter on configuration fuzzing (https://www.fuzzingbook.org/html/ConfigurationFuzzer.html).

a. Implement the argument parser (5 Points)

To get started, familiarize yourself with `argparse`, the Python package for parsing argument line parameters, by reading the documentation at https://docs.python.org/3/library/argparse.html and by studying how it can be used at https://www.fuzzingbook.org/html/ConfigurationFuzzer.html#Options-in-Python .

In this part of the exercise, we ask you to implement an argument parser by implementing a number of calls to `add_argument` and to `add_mutually_exclusive_group`.

Your implementation should go to `exercise_1a.py`, at the position marked with:

```
# TODO: Implement parser.add_argument calls here
```

The parsed arguments will be processed by the existing program logic.

Your argument parser should parse the following arguments:

- An obligatory list of strings, that should be stored to `args.strings`, with `metavar`-name: `s`.
- A mutually exclusive group, consisting of flags for `--identity`, `--sort` and `--reversesort`. Exactly one of the flags *must* be set. The flag that is set will store the constant string `identity`, `sort`, respectively `reversesort` to the constant `args.preprocessing`.
- An optional flag `--output-average-string-length`, which will set the variable `args.output_average_string_length` to `True`, if set.

When running `python3 exercise_1a.py --help`, the output, produced by `argparse`, must be **exactly**:

```
usage: exercise_1a.py [-h] (--identity | --sort | --reversesort) [--output-average-string-length]
s [s ...]

This program concatenates strings.

positional arguments:
  s                     Strings that will be concatenated.
```

```
optional arguments:
  -h, --help            show this help message and exit
  --identity            Do not change individual strings before concatenating.
  --sort                Sort individual strings in normal order before concatenating.
  --reversesort         Sort individual strings in reverse order before concatenating.
  --output-average-string-length
```

This means that you will also have to add explantory help texts as listed above.

b. Mine the EBNF grammar (3 Points)

The next task is to mine a grammar for the argument parser you have implemented in a. For this purpose you should use the `OptionGrammarMiner` class.

Implement the function `mine_ebnf_grammar` in file `exercise_1b.py` at the `#TODO` marker. This function must return the mined grammar.

c. Fuzz with the OptionFuzzer (0 Points)

Feel free to run the program `exercise_1c.py`, which uses the `OptionRunner` and `OptionFuzzer` classes to mine a grammar from the program you have implemented in a., which is then used for fuzzing.

Exercise 11-2: Quiz (2 Points)

In this exercise we will recap the chapters on *Fuzzing Configurations* and *Fuzzing APIs*.

Provide the BEST answers to the following questions in `exercise_2.py` by assigning to each variable `Q1, ..., Q4` the values `1` to `4`.

For instance, if you think the first answer is the BEST answer to Q1, set `Q1=1` in `exercise_2.py`.

There is only **one BEST answer** to each question.

Questions

Q1: Which technique for mining Configuration Options is presented in the lecture?

1. Mining a grammar from command-line documentation.
2. Mining a grammar from command-line samples.
3. Mining a grammar from arbitrary code.
4. Mining a grammar from code using a given arguments-processing convention.

Q2: What is an important drawback of using dynamic analysis for mining Configuration Options?

1. The structure of the command-line processing code needs to be known.
2. If some command-line options are registered in a part of the code that is not executed, they will not be mined.
3. It is impossible to test combinations of options.
4. The type of the options arguments need to be known beforehand.

Q3: Which of the following is an advantage of fuzzing at the API level?

1. It is not necessary to execute the code.
2. There is no need to define a structure (grammar, ...) for the input.
3. It can be much faster than fuzzing at the system level.
4. Several instances of the fuzzer can run in parallel.

Q4: Which of the following is an important drawback of fuzzing at the API level?

1. It is necessary to generate code rather than input data.
2. False alarms can be raised when violating implicit preconditions of the API.
3. It can be much slower than fuzzing at the system level.
4. It is not possible to reuse the grammar fuzzing algorithms used at the system level.