Security Testing
WS 2021/2022

Prof. Dr. Andreas Zeller
Leon Bettscheider
Marius Smytzek

Exercise 5 (10 Points)

Due: 5. December 2021

The lecture is based on The Fuzzing Book (https://fuzzingbook.org/beta), 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 (https://cms.cispa.saarland/fuzzing2122/students/view).

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 5-1: Jump! (6 Points)

In this exercise you will work with coverage results to extract a new modular coverage metric.

Note: To work on this exercise, you will need the latest fuzzingbook version 1.0.1 released on Nov 23, 2021.

To upgrade, use the command `pip3 install fuzzingbook --upgrade`.

a. Linear Code Sequence and Jump (3 Points)

The linear codes sequence and jump coverage metric (https://en.wikipedia.org/wiki/Linear_code_sequence_and_jump) or short *LCSAJ* is a metric that measures all linear sequences of executed lines followed by a jump to another line that is not in sequence. Implement the function `lcsaj()` in **exercise_1a.py** that takes a trace and returns a set of subsequent lines that follow each other until a jump occurs, i.e., a line that is not longer in order. We say that a line is in order with the execution if `current_line_number == previous_line_number + 1`.

The function `lcsaj` should have the signature:

```python
def lcsaj(trace: list[Location]) -> set[tuple[Location, ...]]
```

Note that the type `Location` is defined in the fuzzingbook chapter on Coverage.

Below we show an example of a `lcsaj` run on the function `f` with argument `x=16`.

```python
def f(x):
    if x >= 10:      # L2
        x = x % 10   # L3
                     # L4
    y = x - 5        # L5
    if y < 0:        # L6
        return 0     # L7
    return y         # L8
```

For the following trace

```
[
    ('f', 2),
    ('f', 3),
    ('f', 5),
    ('f', 6),
    ('f', 8),
]
```

the output should look like this: (Note that this is an unordered set)

```
{
    (('f', 2), ('f', 3), ('f', 5)),
    (('f', 5), ('f', 6), ('f', 8)),
    (('f', 8), )
}
```

In this example, there is a jump from L3 to L5, and from L6 to L8. Also, the program exit (L8) is considered a jump.

Note:

- Changing the function (first value of the trace element) is always a jump.
- The end of the program/trace counts as a jump.
- Comments and empty lines increment the line numbers. For simplicity, we treat them as jumps too.
- The jump or better the line after the jump is part of a coverage element.

Tips:

- We would recommend to read **exercise 5-1 b.** before implementing this exercise.
- We use tuples instead of lists, because lists cannot be added to sets. You should work with lists whenever possible, and convert them to tuples when required.
- You are allowed to implement any helper functions you like.
- The function `lcsaj()` needs to return a set but a helper function does not.

b. Make it Modular (2 Points)

Now, you should expand your LCSAJ implementation, such that it takes a parameter `n` and returns a set of `n` consecutive linear code sequences and jumps. You only need to consider `n > 0`. With the example from above the result for `n = 2` should be:

```
{
    ((('f', 2), ('f', 3), ('f', 5)), (('f', 5), ('f', 6), ('f', 8))),
    ((('f', 5), ('f', 6), ('f', 8)), (('f', 8),))
}
```

For `n = 3`:

```
{
    ((('f', 2), ('f', 3), ('f', 5)), (('f', 5), ('f', 6), ('f', 8)), (('f', 8),))
}
```

Only add a consecutive sequence of LCSAJs of length `n` to the output if there are enough LCSAJs. For example for `n = 4` it would then be:

```
{}
```

Implement your solution as the function `lcsaj_n()` in **exercise_1b.py**. The signature should be as follows:

```python
def lcsaj_n(trace: list[Location], n: int) -> set[tuple[Location, ...]]
```

c. Subsumption? (1 Point)

Try to compare LCSAJ-n coverage against line and branch coverage (See Code Coverage Exercise 2 (https://www.fuzzingbook.org/html/Coverage.html#Exercise-2:-Branch-Coverage)). Think about what these metrics measure and when they achieve *100%* coverage. Then answer the following questions. These are general questions, i.e. your answers should hold for all possible programs. Provide your solution in **exercise_1c.py**, assign each corresponding variable `Q1` to `Q4` `True`, if you think the answer is **yes**, and `False`, if you think the answer is **no**.

Q1: Do you always achieve *100%* statement coverage when achieving *100%* LCSAJ-1 coverage?

Q2: Do you always achieve *100%* branch coverage when achieving *100%* LCSAJ-1 coverage?

Q3: Do you always achieve *100%* LCSAJ-1 coverage when achieving *100%* statement coverage?

Q4: Do you always achieve *100%* LCSAJ-1 coverage when achieving *100%* branch coverage?

Exercise 5-2: Leave on a High Note (4 Points)

We are getting back to statement coverage as introduced in Code Coverage (https://www.fuzzingbook.org/html/Coverage.html). For this exercise you will work on using statement coverage feedback to decide when to stop fuzzing. In the following you will extend the `RandomFuzzer` class from the chapter Fuzzing: Breaking Things with Random Inputs (https://www.fuzzingbook.org/beta/html/Fuzzer.html), such that it takes a runner measuring the coverage of a given function and stops creating new inputs when no change in the coverage can be detected.

a. Put the Fun in Functions (but this Time with Coverage) (1 Point)

In this exercise you should implement a runner that executes a function and measures its coverage during the execution, as described in chapter Code Coverage (https://www.fuzzingbook.org/html/Coverage.html#A-Coverage-Class). Please implement the class `FunctionCoverageRunner` in **exercise_2a.py** by overriding the `run_function()` method. When implementing this class take care of the following points:

- If the function raises an exception you should still be able to provide the coverage.
- The coverage of the function should be accessible after calling the run method.

b. Stop the Fuzzer (3 Points)

Now, you should implement the new fuzzer `RandomCoverageFuzzer`. The fuzzer inherits from the `RandomFuzzer`. Please override the `runs()` method, such that it stops as soon as there is no significant change in the total coverage. To accomplish this you should keep track of the total coverage and stop the fuzzing when **10** consecutive iterations of the fuzzing, i.e. produce a fuzzed input and execute it (you can execute this by calling `self.run(runner)`), did not cover any previously uncovered lines. As soon as a new line is covered, reset the count of iterations, such that only consecutive iterations are considered.