CISPA HELMHOLTZ CENTER FOR INFORMATION SECURITY

Security Testing
WS 2023/2024

Prof. Dr. Andreas Zeller
Leon Bettscheider
José Antonio Zamudio Amaya

Exercise 9 (10 Points)

Due: 14. January 2024 23:59

> 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 9-1: Taint me like one of your French girls (7 Points)

In this exercise you will implement a type independent tainting. This will show you how flexible and powerful tainting in python could be. All your solutions for this exercise must be implemented in **exercise_1.py**. In this exercise you will always have the opportunity to validate your solutions by executing

```
python3 tests_1.py
```

This script comprises *14* tests. For each passed test, you will receive *0.5* points. The printed information tells you if you passed a test and where your implementation failed.

All functions that should be added are a member of one of the variables in the top area of **exercise_1.py**.

a. Setup

Implement the `create(self, x: Any)` method in the `tany` class that returns a new `tany` object with the value `x` and that taint `self.taint`.

b. Operation support

Implement the following functions that return a tainted object:

In [ ]:
```
ops = ['__add__', '__sub__', '__mul__', '__matmul__', '__truediv__',
       '__floordiv__', '__mod__', '__divmod__', '__pow__', '__lshift__',
       '__rshift__', '__and__', '__xor__', '__or__', '__lt__', '__le__',
       '__gt__', '__ge__','__eq__', '__ne__']
```

And there right sided counterparts:

In [ ]:
```
rops = ['__radd__', '__rsub__', '__rmul__', '__rmatmul__', '__rtruediv__',
        '__rfloordiv__', '__rmod__', '__rdivmod__', '__rpow__',
        '__rlshift__', '__rrshift__', '__rand__', '__rxor__', '__ror__',
        '__rlt__', '__rle__', '__rgt__', '__rge__','__req__', '__rne__']
```

You can leverage the `make_wrapper(fun_name)` function to add all methods. Keep in mind that `other` could already be a `tany` object. If you use `make_wrapper(fun_name)`, the function already takes care of this.

To add a function you could do the following:

```
setattr(tany, '__add__', tany.make_wrapper('__add__'))
```

c. Number support

Next, you need to add the leftover functions to support numbers:

In [ ]:
```
number_ops = ['__neg__', '__pos__', '__abs__', '__invert__',
              '__round__', '__int__', '__float__', '__complex__']
```

You can again leverage `make_wrapper(fun_name)`.

Then you need to add the casting of types, this is done by calling the already implemented `tint(x)` and `tfloat(x)`. Add a new function `tcomplex(x)` that follows the same structure as the other two but with `complex(x)` instead of `int(x)` or `float(x)`.

d. String support

Next, you need to add the leftover functions to support strings:

In [ ]:
```
str_ops = ['__repr__', '__str__', 'capitalize', 'casefold', 'center',
           'encode', 'expandtabs', 'format', 'format_map', 'join',
           'ljust', 'lower', 'lstrip', 'replace', 'rjust', 'rstrip',
           'strip', 'swapcase', 'title', 'translate', 'upper',
           'startswith', 'split', 'isupper', 'isspace', 'istitle',
           'isprintable', 'isnumeric', 'islower', 'isidentifier',
           'isdigit', 'isdecimal', 'isalpha', 'isalnum']
```

You can again leverage `make_wrapper(fun_name)`.

e. Container and Iterator support

Finally, you need to add the support for containers like `list` and iterators with the following functions:

In [ ]:
```
container_ops = ['__len__', '__getitem__', '__iter__', '__next__',
                 '__reversed__', '__missing__', 'append', 'extend',
                 'insert', 'remove', 'pop', 'clear', 'index',
                 'count', 'sort', 'reverse', 'copy']
```

You can again leverage `make_wrapper(fun_name)`.

There are some functions here that we cannot alter because they are passed through the non accessible core of python but we need to include them to provide support.

In [ ]:
```
container_ops_unchanged = ['__setitem__', '__delitem__', '__contains__']
```

You can leverage `make_wrapper_unchanged_return(fun_name)` that does not alter the return value of these functions. You can use it the same way as `make_wrapper(fun_name)` before. Keep in mind that those functions cut off the tainting when using the return value.

Exercise 9-2: Quiz (3 Points)

In this exercise we will recap different aspects of taint tracking.

Provide the BEST answers to the following questions in **exercise_2.py** by assigning to each variable `Q1, ..., Q3` 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: What is the core idea of taint tracking?

1. Taint tracking makes the program run faster.
2. Finding a solution to a conditional branch with an SMT solver.
3. Instrumenting the code at the binary level.
4. Finding out whether there is a leak from a source to a sink.

Q2: Which information flow (into `a`) is hard to track?

1. a = my_tainted_variable
2. a = my_tainted_variable * 2
3. if my_tainted_variable == 42: a = 42
4. for v in possibly_tainted_variables: a += v

Q3: How can taint tracking improve fuzzing?

1. If, via taint tracking, we know that only a small part of the input flows into an interesting or dangerous part of the code, we can focus the fuzzer only on this small part of the input.
2. It is especially easy to implement across large software stacks, consisting of multiple operating systems, protocols, and applications.
3. Taint tracking precomputes all possible values for each byte in the input, so the fuzzer has to do less work.
4. Taint tracking is superior to fuzzing, so we can replace fuzzing completely with taint tracking.