CISPA HELMHOLTZ CENTER FOR INFORMATION SECURITY

Security Testing
WS 2021/2022

Prof. Dr. Andreas Zeller
Leon Bettscheider
Marius Smytzek

Exercise 9 (10 Points)

Due: 16. 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.

Please update the `fuzzingbook` package for this exercise with

```
pip3 install --upgrade fuzzingbook
```

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

```
python3.9 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 [4]:
```
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 [5]:
```
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 [7]:
```
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 [10]:
```
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 [12]:
```
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 [13]:
```
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: Follow the Trace (3 Points)

In this exercise you will work with concolic tracing and fuzzing.

a. Tracing (2 Points)

Implement the function `run(func: callable, args: tuple)` in **exercise_2a.py**. The function contains 3 *TODOs* where you need to add code:

1. Set up a `ConcolicTracer` object called `ct` and use it to execute the function `func` on the arguments `args`. You can execute a function `f` on a tuple `t` by calling `f(*t)`.
2. Evaluate the path constraints collected with `ct` by calling the correct function.
3. Set up a `ConcolicTracer` object called `ct2` and use it to execute the function `func` on the arguments `args2`. You can execute a function `f` on a list `l` by calling `f(*l)`.

The first part of the output when running `python3.9 exercise_2a.py` should look like this:

```
[Or(1 < sigma_n_int_2, 1 == sigma_n_int_2), Or(2 < sigma_n_int_2, 2 == sigma_n_int_2), Or(3 <
sigma_n_int_2, 3 == sigma_n_int_2), Or(4 < sigma_n_int_2, 4 == sigma_n_int_2), Not(Or(5 <
sigma_n_int_2, 5 == sigma_n_int_2))]
('sat', {'x': ('0', 'Int'), 'n': ('4', 'Int')})
[Or(1 < sigma_n_int_2, 1 == sigma_n_int_2), Or(2 < sigma_n_int_2, 2 == sigma_n_int_2), Or(3 <
sigma_n_int_2, 3 == sigma_n_int_2), Or(4 < sigma_n_int_2, 4 == sigma_n_int_2), Not(Or(5 <
sigma_n_int_2, 5 == sigma_n_int_2))]
```

b. Fuzzing (1 Point)

Implement the *TODOs* in **exercise_2b.py** to finish the fuzzer:

1. Add the trace to `scf` by calling the correct function with ct and '10' as arguments.
2. Fuzz a new string from `scf`.
3. Call the `totient()` function on `v` under the `ConcolicTracer` `ct`. Keep in mind that `v` is a string but totient requires an integer, which means that you need to convert `v` before handing it over to `totient`.
4. Add the trace to `scf` by calling the correct function with ct and `v` as arguments.

The fuzzing should produce the following output when executing `python3.9 exercise_2b.py`:

```
12
0
7
2
4
5
4
7
3
5
```