CISPA HELMHOLTZ CENTER FOR INFORMATION SECURITY

# Security Testing
WS 2023/2024

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

## Exercise 2 (10 Points)

Due: 19 November 2023 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.

You must verify that your submission is valid by running:

```
python3 verify.py
```

The output tells you whether your submission meets our structural expectations, and gives a warning in case a required file, variable, or function is missing. If you do not follow this structure or change it, we cannot evaluate your submission. **In this case, we will grade your submission with 0 points.** Note that the script does not check if your solutions are correct.

---

**In this exercise sheet, you will write context-free grammars for two programming languages.**

## Exercise 2-1: (Brainf*ck) (5 Points)

Please familiarize yourself with the brainf*ck programming languages by reading the wikipedia page. In particular, focus on the eight commands of the programming language:

```
> < + - . , [ ]
```

### a. Write a grammar (2 Points)

Please write a context-free grammar in fuzzingbook format for the brainf*ck programming language.

The start symbol must be `<start>`.

This grammar should be able to produce the set of all brainf*ck programs.

Make sure that the programs produced by your grammar do have balanced parentheses.

Store the grammar in **bf_grammar.py**. The variable name of the grammar must be `BFGRAMMAR`.

Before submitting, please make sure your grammar is valid by running `assert is_valid_grammar(BFGRAMMAR)`, or, alternatively run the `verify.py` program which also performs this check.

### b. Just fuzz it (3 Points)

We implemented a brainf*ck interpreter in **bf.py**. Unfortunately, we felt a bit dizzy during programming and introduced **4** bugs (1 bug per line) into the program.

Please use **fuzzBF.py** with the grammar you've written in `Exercise 2-1a` to find those bugs using fuzzing. Logs are written to stdout and help you to diagnose the errors.

It is sufficient to run the fuzzer **1000** times to solve this exercise.

### b-1. Where do errors become apparent? (1 Point)

List all the lines in **bf.py** where an error occured during fuzzing. Note that the line where an error occured is not necessarily the line where it originated from (root cause).

**Note**: `timeout` does not indicate a bug. Timeouts will happen as the grammar can generate programs that are not guaranteed to terminate.

Provide your solution in **exercise_1b1.py**. For instance, if you think an error occured in lines 1,3,6,7,8 your **exercise_1b1.py** file should look as follows:

In [ ]:
```
lines = [1,3,6,7,8]
```

Note that you may not name more than 7 lines.

### b-2. Where do the errors originate? (2 Points)

All the errors that can be observed during fuzzing originate from **4** faulty lines of code.

Please give the line numbers of the **4** faulty lines in **bf.py** and explain the fault for each line (one sentence each).

Provide your solution in **exercise_1b2.py**. Structurally, your solution should look as follows:

In [ ]:
```
line1 = 111
line2 = 222
line3 = 333
line4 = 444
explanation_line1 = "The bug occured due to an ZZZ error: Instead of XXX, tl
explanation_line2 = "The bug occured due to an ZZZ error: Instead of XXX, tl
explanation_line3 = "The bug occured due to an ZZZ error: Instead of XXX, tl
explanation_line4 = "The bug occured due to an ZZZ error: Instead of XXX, tl
```

## Exercise 2-2: TinyC (3 Points)

TinyC is a simplified subset of the C programming language.

The TinyC language is characterized by the following context-free grammar in BNF notation:

```
<program> ::= <statement>
<statement> ::= "if" <paren_expr> <statement> |
                "if" <paren_expr> <statement> "else" <statement> |
                "while" <paren_expr> <statement> |
                "do" <statement> "while" <paren_expr> ";" |
```

```
                "{" <statement>* "}" |
                <expr> ";" |
                ";"
<paren_expr> ::= "(" <expr> ")"
<expr> ::= <test> | <id> "=" <expr>
<test> ::= <sum> | <sum> "<" <sum>
<sum> ::= <term> | <sum> "+" <term> | <sum> "-" <term>
<term> ::= <id> | <int> | <paren_expr>
<id> ::= "a" | "b" | "c" | "d" | ... | "z"
<int> ::= <an_unsigned_decimal_integer>
```

Translate the grammar given above to fuzzingbook syntax. The start symbol should be `<start>`.

Store the grammar in **tinyc_grammar.py**. The grammar's variable name should be `TINYCGRAMMAR`.

Before submitting, please make sure your grammar is valid by running `assert is_valid_grammar(TINYCGRAMMAR)`, or, alternatively run the `verify.py` program which also performs this check.

**Optional**: To see your grammar in action, first compile the TinyC compiler (**tinyc.c**) using your favorite C compiler, e.g. `gcc tinyc.c -o tinyc`. The resulting executable should be named **tinyc**. Next, run **fuzzTinyC.py** and marvel at the programs your grammar generates.

## Exercise 2-3: Quiz (2 Points)

Provide the BEST answers to the following questions in `exercise_3.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_3.py`.

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

Consider the following grammar in BNF syntax:

```
<start>   ::= <number>
<number>  ::= <integer> | +<integer> | -<integer>
<integer> ::= <digit> | <digit><integer>
<digit>   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

**Q1**: Which statement is correct w.r.t. the grammar given above?

1. The grammar is not a context-free grammar.
2. The grammar describes all floating point numbers.
3. If a parse tree for a derivation of the given grammar contains a node `1`, this node must be a leaf node.
4. If a parse tree for a derivation of the given grammar contains a node `<digit>`, this node can be both a leaf node or an inner node.

**Q2**: Which statement is correct w.r.t. the grammar given above?

1. `+<integer>` is a non-terminal symbol.
2. `<digit>` is a non-terminal symbol.
3. `1` is a non-terminal symbol.
4. Both `+<integer>` and `<digit>` are non-terminal symbols.

**Q3**: Which statement is correct w.r.t. the grammar given above?

1. `<start>` is a rule of some non-terminal symbol.
2. `<digit><integer>` is a rule of some non-terminal symbol.
3. `<digit><number>` is a rule of some non-terminal symbol.

4. `111` is a rule of some non-terminal symbol.

**Q4**: Which statement is correct?

1. All programming languages can be described by context-free grammars.
2. Only programming languages with a context-free syntax can be described by context-free grammars.
3. The C++ programming language can be described by a context-free grammar.
4. The Python programming language can be described by a context-free grammar.