LOG6305 - Techniques avancées de test du logiciel

Assignment 2

Fuzzing for test generation

Département de génie informatique et de génie logiciel

École Polytechnique de Montréal

Hiver 2024

1 Introduction

Fuzz testing, also known as fuzzing, is an automated software testing method that injects invalid, malformed, or unexpected inputs into a system to reveal software defects and vulnerabilities. In this assignment, you will conduct fuzz testing on the three suggested systems and explore techniques to perform fuzzing more efficiently.

2 Objectives

The objectives of this work are: :

1. Be able to fuzz a given program with random inputs, represented as strings.
2. Be able to use various criteria and strategies to guide the fuzzer in selecting more relevant inputs.
3. Be able to add a grammar to your fuzzer to generate semantically correct inputs.

3 Introduction to fuzzing

According to whether the tester knows the structure of the program or not, there can be different types of fuzzing:

- **Black Box Fuzzing:**Treats the program as a black box and ignores the internal structure of the program. For example, a random testing tool that generates random inputs is considered a black box fuzzer.
- **White Box Fuzzing:** Relies on program analysis to systematically increase code coverage or to reach certain critical locations within the program. For instance, one may execute the program (starting with a few random inputs), gather constraints on inputs at conditional statements, use a constraint solver to generate new test inputs.
- **Grey Box Fuzzing:** Tests the program with partial knowledge of its internal workings. For example, a grey box fuzzer might leverage coverage feedback from other instrumentation or libraries to learn how to dive deeper into the program. If a generated input increases coverage, it will be learned by the fuzzer to further enhance fuzzing.

In this assignment, we will focus on grey box fuzzing, as it is most commonly used in practice. Below we present some key concepts and ideas of grey box fuzzing.

Mutation-based fuzzing. In the very simple case, fuzzing assumes generating random inputs with the aim to falsify a given program. However, most randomly generated inputs are syntactically invalid and thus are quickly rejected by the processing program. To exercise functionality beyond input processing, we must increase chances to obtain valid inputs. One such way is so-called mutational fuzzing – that is, introducing small changes to existing inputs that may still keep the input valid, yet exercise new behavior. A mutation in this context is a simple string manipulation - say, inserting a (random) character, deleting a character, or flipping a bit in a character representation. This is called mutational fuzzing – in contrast to the generational fuzzing techniques discussed earlier. For more detailed information we recommend the 'Mutation-Based Fuzzing' [1] chapter from the 'Fuzzing book' [1].

Coverage-guided fuzzing The main idea of coverage-guided fuzzing is to prioritize mutated inputs that increase certain code coverage criteria, such as line coverage. This approach allows us to focus on mutating inputs that have so far uncovered new paths in the program, increasing the likelihood of discovering inputs that cover even larger portions of the program.

Power schedule A power schedule distributes the precious fuzzing time among the seeds in the population. Our objective is to maximize the time spent fuzzing those (most progressive) seeds which lead to higher coverage increase in shorter time.

We call the likelihood with which a seed is chosen from the population as the seed's energy. Throughout a fuzzing campaign, we would like to prioritize seeds that are more promising. Simply said, we do not want to waste energy fuzzing non-progressive seeds. We call the procedure that decides a seed's energy as the fuzzer's power schedule. For instance, AFL's schedule assigns more energy to seeds that are shorter, that execute faster, and yield coverage increases more often. For more detailed information on coverage-guided fuzzing, we refer you to the Greybox Fuzzing chapter[2] of the 'Fuzzing book'.

Fuzzing with grammars Specifying inputs via a grammar allows for very systematic and efficient test generation, in particular for complex input formats. Grammars are among the most popular (and best understood) formalisms to formally specify input languages. Using a grammar, one can express a wide range of the properties of an input language. Grammars are particularly great for expressing the syntactical structure of an input, and are the formalism of choice to express nested or recursive inputs. More detailed information is provided in the 'Fuzzing book' [3].

4 The tasks

Required set-up

Operational system : Linux/macOS/Windows, **Python** : version 3.10.

Obtain a local copy of the code for the first practical work, located in the github repository: `https://github.com/log6305/HIV_2024_TP2`. We suggest to create the fork of this repository and make changes to your fork (later you could send a link to your fork for evaluation).

This repository contains a baseline implementation of the Mutation fuzzers you are required to develop in this assignment. Complete the provided the jupyter notebook to better understand how to use the proided code.

[1]`https://www.fuzzingbook.org/html/MutationFuzzer.html`

[2]`https://www.fuzzingbook.org/html/GreyboxFuzzer.html`

[3]`https://www.fuzzingbook.org/html/Grammars.html`

Your tasks:

Your main task is to create 3 Fuzzers for the three proposed python modules.

1. Question 1: Create a Mutation fuzzer (child of AbstractFuzzer class) to test the ‘cgi_decode.py’ function provided in the repository. You can use an evaluation budget of no more than 100. You should create mutations for the inputs of your choice (2 points), define the input grammar (2 points) as well as the power schedule to prioritize better seeds (3 points). You should compare three versions of your implementation: implementation including power schedule, implementation when the power schedule is not used (and only coverage guides the test generation) and implementation with power schedule, but no grammar. No more than 10 seeds can be used (3 points). In your answer, provide the achieved coverage convergence plots based on minimum 10 runs. Save your fuzzer in the ‘$poly_{f}uzzer/fuzzers/cgi\_fuzzer.py$’ file and the power schedule in the ‘*poly_fuzzer/power_schedules/url_schedule.py*’ file.

2. Question 2: Create a Mutation fuzzer (child of AbstractFuzzer class) to test the ‘urlparse’ function from the ‘urllib.parse’ module. You can use an evaluation budget of no more than 200. You should create mutations for the inputs of your choice (3 points), define the input grammar (3 points) as well as the power schedule to prioritize better seeds (4 points). You should compare three versions of your implementation: implementation including power schedule, implementation when the power schedule is not used (only coverage guides the test generation) and implementation with power schedule, but no grammar No more than 10 seeds can be used (3 points). In your answer, provide the achieved coverage convergence plots based on minimum 10 runs. Save your fuzzer in the ‘*poly_fuzzer/fuzzers/url_fuzzer.py*’ file and the power schedule in the ‘*poly_fuzzer/power_schedules/url_shcedule.py*’ file.

3. Question 3: Create a Mutation fuzzer (child of AbstractFuzzer class) to test to test the ‘HTMLParser().feed’ function from the ‘html.parser’ module. You can use an evaluation budget of no more than 500. You should create mutations for the inputs of your choice (4 points), define the input grammar (4 points) as well as the power schedule to prioritize better seeds (5 points). You should compare three versions of your implementation: implementation including power schedule, implementation when the power schedule is not used (and only coverage guides the test generation) and implementation with power schedule, but no grammar (no more than 10 manually specified seeds can be used) (3 points). In your answer, provide the achieved coverage convergence plots based on minimum 10 runs. Save your fuzzer in the ‘*poly_fuzzer/fuzzers/url_fuzzer.py*’ file and the power schedule in the ‘*poly_fuzzer/power_schedules/url_shcedule.py*’ file.

4. Question 4: Select the best best configuration (the one ahcieving the highest line coverage on average) for each of the fuzzers you developed. Report the best maximal coverage achieved based on 10 runs. Respect the indicated budget as well as the number of manually specified seeds (no more than 10). You will be given additional points based on the average performance of your fuzzer across the three modules under test. Your additional points will be proportional to the ranking. The maximum additional points you can achieve equal to 5% of the assignment grade.

Important instructions:

1. If possible, do not modify the implementation of the Abstract classes. You can refer to the Mutation Fuzzer class as an example to develop your Fuzzer, but you are free to implement it as you want.
2. In your report, provide the description of your Fuzzer for each of the questions. For questions 1-3 describe the Fuzzer you implemented and discuss the evaluation results. For presenting results, use plots and tables (you should add the convergence plots i.e. the number of lines covered vs the number of inputs used). In the question 4 explain what configurations did you choose and why.

5 Expected deliverables

The following deliverables are expected:

- Link to your repository with the implemented fuzzers or a .zip archive of the repository.
- The report for this practical work in the PDF format.

You should submit all the files to Moodle. When adding the PDF file, do not put into a zip. Additionally, you should add a ".txt" file with the following content: "Your full name, your student id" (this might be used for the grading automation). This assignment is individual.

The report file must contain the title and number of the laboratory, your name and student id.

6 Important information

1. Check the course Moodle site for the file submission deadline
2. A delay of [0.24 hours] will be penalized by 10%, [24 hours, 48 hours] by 20% and more than 48 hours by 50%.
3. No plagiarism is tolerated (including ChatGPT). You should only submit code created by you.

References

[1] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2024. Retrieved 2024-01-18 17:28:37+01:00.