UNIVERSITY OF ANTWERP
DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE

Software Testing Lab

Assignment 8

Submission Deadline: **May 8th, 20:00**

Refer to Assignment 1 for introductory information.
Note that this assignment has a 25 point late fee instead of additional exercises.

FUZZING

Let's start with a short story from Prof. Barton Miller:

> It was a dark and stormy night. Really.
>
> Sitting in my apartment in Madison in the Fall of 1988, there was a wild midwest thunderstorm pouring rain and lighting up the late night sky. That night, I was logged on to the Unix system in my office via a dial-up phone line over a 1200 baud modem. With the heavy rain, there was noise on the line and that noise was interfering with my ability to type sensible commands to the shell and programs that I was running. It was a race to type an input line before the noise overwhelmed the command.
>
> This fighting with the noisy phone line was not surprising. After all, this was just before error-correcting modems were available. What did surprise me was the fact that the noise seemed to be causing programs to crash. And more surprising

> to me were the programs that were crashing – common Unix utilities that we all use everyday.
>
> The scientist in me said that we need to make a systematic investigation to try to understand the extent of the problem and the cause.
>
> That semester, I was teaching the graduate Advanced Operating Systems course at the University of Wisconsin. Each semester in this course, we hand out a list of suggested topics for the students to explore for their course project. I added this testing project to the list.
>
> In the process of writing the project description, I needed to give this kind of testing a name. I wanted a name that would evoke the feeling of random, unstructured data. After trying out several ideas, I settled on the term fuzz. [1]

Fuzzing or fuzz testing is an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program. The program is then monitored for exceptions such as crashes, failing built-in code assertions, or potential memory leaks.

In this assignment, you will write a black-box fuzzer to fuzz JPacman. Since you will do a black-box fuzzing, you are provided an executable jar file and a sample map file. In this version of JPacman we have some new updates: (1) A bug is injected in the code, (2) it loads map files (3) it replays an action sequence.

The jar file can be executed in this way:

```
$ java -jar jpacman-3.0.1.jar awesome.map SUUWDE
```

It loads the map (awesome.map) and then replays the action sequence (SUUWDE). In this case, it starts the game (S), moves the players up twice (UU), waits (W), moves down (D), and finally, it exits the game (E). You can find a complete list of actions in table 1. In this table p is an instance of the class `Pacman` initiated in the main method. Other characters in the action sequence will be ignored.

Key	Action	Key	Action
E	p.exit()	**Q**	p.quit()
S	p.start()	**W**	Thread.sleep(50)
U	p.up()	**L**	p.left()
D	p.down()	**R**	p.right()

Table 1: Possible actions

For simplicity, the jar file uses different exit codes:

- Exit code 0: a normal termination.
- Exit code 10: a rejection. The application is able to handle this invalid input.

[1] http://pages.cs.wisc.edu/ bart/fuzz/Foreword1.html

- Exit code 1: a crash. The application is not able to handle this input.

- **Exercise 1.** Write a fuzzer in a language you like (Java, Python, ...). Your fuzzer will do these steps in a loop: (1) generating an input file and a random action sequence (2) running JPacman with generated inputs (3) checking the exit code of the program. Your fuzzer can exit the loop after a maximum iteration (MAX) or when a time budget finishes (TIME). For the maps, in this step, your fuzzer only generates random binary files. Send your fuzzer code. **(Required, 20 points)**

- **Exercise 2.** Run your fuzzer, which generates pure random inputs (MAX=1000). What do you see? **(Required, 5 points)**

- **Exercise 3.** Do you think a completely random fuzzing is efficient? Why? **(Required, 5 points)**

Let's do some manual fuzzing and try to guess what is going on behind the scene based on the received errors. We can use this information to make the fuzzer more efficient.

As an example, when we use a binary file, we see the error "*Error reading file by readAllLines*". We can infer the program rejects all non-text files by this error. Then we provide a text file with multiple lines. We see the error changes to "*Widths mismatch*".

Based on these errors we can guess some part of the code in the system under test and build pseudocode like this[2] (pay attention to the order of the checks):

```
if(file is not a text file)
   reject(Error reading file by readAllLines);
if(line lengths are not identical)
   reject(Widths mismatch);
```

- **Exercise 4.** Continue manual fuzzing. Try corner cases like an empty file, empty lines, special characters, and .... Find as many errors as you can and their order. Describe the activities you performed and write a pseudocode. **(Required, 20 points)**

- **Exercise 5.** Based on the information you have gathered by manual fuzzing, improve your fuzzer. Run your fuzzer to fuzz JPacman (TIME= 10 minutes). Send your fuzzer code and the result (only crashes). **(Required, 20 points)**

Manual fuzzing needs human interaction and it depends on tester experiences. An alternative technique to make the fuzzer more efficient is Mutation based fuzzing. In this technique, the fuzzer will use a valid and well defined input and mutate some parts of that. Inputs generated by this method are more likely to pass initial checks and test the program deeply.

- **Exercise 6.** Change your fuzzer to a mutation-based one. Run your fuzzer on JPacman (TIME= 10 minutes). Send your fuzzer code and the result (only crashes). **(Required, 30 points)**

[2] This technique is also useful in web-applications penetration testing.

- **Exercise 7.** Watch this video https://www.youtube.com/watch?v=1S0aBV-Waeo about the buffer overflow attack in C. How can fuzzing help a security tester to find buffer overflow vulnerabilities? **(Bonus, 10 points)**

- **Exercise 8.** Watch the video or read the article about SAGE, a white-box fuzzing technique, and answer these questions:
  - Video: https://channel9.msdn.com/blogs/peli/automated-whitebox-fuzz-testing-with-sage
  - Article: https://patricegodefroid.github.io/public_psfiles/cacm2012.pdf

  (1) What is the limitation of black-box fuzzing?
  (2) How does SAGE solve this problem?
  (Bonus, 15 points)

Late fee: -25 points