Advanced Software Testing and Reverse Engineering Lab 1

INTRODUCTION

Software contains many errors that are difficult to find. Methods for locating such errors come in two flavors: static and dynamic analysis. Static analysis searches the code for patterns that indicate the presence of errors such as a use-after-free vulnerability. Dynamic analysis instead executes the code, monitors this execution, and aims to find inputs that trigger vulnerabilities such as segmentation faults. This course gives an overview of different techniques for dynamic analysis, which can be divided into three categories:

- *Black-box – the code or internal software structure is not used*
- *Grey-box – uses parts of the internal structure/code as guidance*
- *White-box – fully interprets the internal structure/code*

*The first lab focusses on **grey-box methods**. You will start by building a simple **black-box fuzzer** that continuously probes a program with random input, hoping to cause problems. You will then improve this fuzzer by guiding it using **grey-box** details such as the branch coverage, a flow analysis, or input taints. Finally, your task is to compare your fuzzer to the state-of-the-art. The programs that we will use throughout this class come from the 2019 RERS (reverse engineering of reactive systems) challenge.*

- *http://rers-challenge.org*

These are highly obfuscated pieces of software that take strings as input and return strings as output. On Brightspace, you can download a Docker container containing the relevant problems, as well as a multitude of programs that you will use throughout this course.

LEARNING OUTCOMES

After completing this assignment, you will be able to:

- *Build fuzzers from scratch.*
- *Instrument Java applications to gather grey-box information.*
- *Use grey-box information to build smarter fuzzers that use search.*
- *Use and understand AFL (http://lcamtuf.coredump.cx/afl/).*
- *Analyze differences between fuzzing results.*

INSTRUCTIONS

Read the content on Brightspace on fuzzing, tainting, branch distance, and hill-climbing.

Download the Docker container from Brightspace.

The RERS problems are in the RERS directory. There is a small Python script available in the home directory that can be used to add instrumentation to the RERS Java files. You can have a look by opening it in an editor, it uses regular expressions to replace parts of the Java file and creates assembly-like statements that are still Java code. Run it on each of the RERS training problems to create Java files starting with the word inst. Have a look at one of the instrumented files, compare it with the original, and understand the difference. In reality, such instrumentated code will be executed on bytecode, making it much harder to observe these differences.

A simple random fuzzer is already implemented. Compiling and executing the instrumented Java file should start this fuzzer. Your job is to make this fuzzer smarter.

Task 1: tainting

Tainting is the process of tracking user input through programs. Using the instrumentation, you can track user input (generated by the fuzzer). Expand the instrumented java code (either in the code itself, or in the Python script, such that it tracks taints. Notice the MyString class already contains a Boolean flow variable, which is set to true by the fuzzer. You should keep track of this flow, for instance by keeping track of all assignments, or by maintaining a set of tainted variables, or by simply propagating these Boolean values. Note that conditional statements and subsequent assignments may also propagate flow:

```
if(a.equals("A")) {b = "A";}
if(a.equals("B")) {b = "B";}
..
```

In other words, whenever the value of a variable depends on a tainted variable, it should become tainted as well. It is quite difficult to discover whether flow should be propagated after conditional statements. You should implement two versions: one that always propagates after conditional statements, and one that only propagates in direct assignments.

Test and compare your taints on the RERS training problems. Show how far the taints propagate after varying lengths of input traces.

Task 2: branch distance

Similar to the tainting task, you have to update the instrumentation to compute the code branches it reaches and the current branch distance. Build data structure for storing the visited code branches, make sure the lookup is efficient, specifically, we will use it to:

- Lookup whether we already visited the branch for a given input trace
- Compute which if-condition needs to be triggered to visit a new branch

Branch distance is used to estimate how far the current input trace is from triggering the next code branch. You can compute this in the MyIf statements, but you have to know what condition is checked by the MyBool variable it tests. Build a method for doing so, for instance by tracking all Boolean assignments, or by monitoring and storing the most recent Boolean tests. How you do this is up to you. Store the branch distance for every input trace in the data structure.

Display the reached code branches including branch distances for a set of input traces on the RERS training problems.

Task 3: search

We can use the computed taints and branch distances to guide the fuzzer. How to do this is entirely up to you. Ideas:

- Compute the branch distance for a current trace, try random permutations, pick one that lowers this distance, and iterate.
- Pick a trace from the visited branches with a large taint, randomly permute this trace, iterate.
- Randomly restart the fuzzer from a trace with large taint/low distance.
- …

Implement a simple search strategy and try to reach all code branches for the RERS training problems. Try some of the reachability problems and keep track of which errors you can reach.

Task 4: AFL

You are asked to apply AFL to the RERS training and reachability problems. AFL is a state-of-the-art fuzzer that uses genetic algorithms and branch coverage for guidance. Instructions on how to set up AFL for the RERS problems are available on Brightspace.

Compare the performance of AFL with your own fuzzer. The findings/crashes directory contains all crashes AFL found, in this case there are the reachability errors. By running these through the normal (uninstrumented) code, you can obtain all the found reachability statements. Does AFL reach more reachability statements than your own fuzzer? What about branches? Investigate the examples found by AFL. Did you also test these? If not, can you improve your search to try to find them?

RESOURCES

The lab is run in a docker container, available at:

https://surfdrive.surf.nl/files/index.php/s/qa9m0nKeRkWZcel/download?path=%2F&files=str.tar

For the first lab, you will need the RERS problems, see: http://rers-challenge.org/2019/. These are already in the container.

You also need to run a small script called regex.py on these problems, available in the homedir.
You should read and understand the fuzzingbook chapter on mutation-based fuzzing:

https://www.fuzzingbook.org/html/MutationFuzzer.html

PRODUCTS

A small report of max 2 A4 pages answering the questions from the 4 tasks, visualizations are allowed in at most 2 extra A4 pages.

A archive (tar/zip) containing the code for computing the results.

ASSESSMENT CRITERIA

You can either pass or fail this assignment. We expect everyone to pass. You have to complete all lab assignments. Submissions of which the report text does not fit into 2 A4s will not be evaluated.

Your work will be evaluated based on its completeness (having done all tasks), the correctness of the implementation, and demonstration that you understand the results in the analysis.

When deemed insufficient, you will receive feedback and will be given a one-week grace period to fix any shortcomings.

SUPERVISION AND HELP

There will be lab sessions every Wednesday, where the teachers and TA will be available to answer any questions you may have. The preferred way to ask questions is through Mattermost.

SUBMISSION AND FEEDBACK

The submission is through Brightspace. You will receive feedback within one week after the deadline.