Ira A. Fulton Schools of Engineering
Arizona State University

CSE 543: Information Assurance and Security

Fuzz them all

Purpose

The purpose of this assignment is to test your understanding of fuzz testing (fuzzing) and guide you to develop your own binary fuzzer. You will learn how to develop your own simple fuzzer, monitor for crashes, generate high-quality seed input, etc.

Objectives

Students will be able to:

- Determine how fuzzers for binary programs work.
- Differentiate between types of fuzzers
- Categorize advantages of each type of fuzzer (e.g., smart fuzzers vs. dumb fuzzers and mutational vs. generative fuzzers).
- Understand the importance of seed input selection.
- Develop a simple fuzzer to fuzz binary programs when source code is available.
- Develop a simple fuzzer to fuzz binary programs when source code is not available.

Technology Requirements

You may use any programming language to implement the fuzzer. However, for a better speed, native languages, such as C, C++, Go, and Rust are strongly recommended. You may also use Java or C# with some acceptable level of speed penalty compared to native languages. You are not encouraged to use Python, but you can if you wish.

Your fuzzer will work against Linux programs that accept input through stdin and generate output to stdout. Windows or MacOS programs are not valid targets. If you are using Windows or MacOS, you may want to install a virtual machine application. You can get VMware for free as an ASU student (strongly recommended) and install Ubuntu 20.04 LTS in a VM.

Project Description

For this assignment, you are supposed to develop a dumb mutative fuzzer. You may want to develop a smart fuzzer. However, it is important to know that developing a good fuzzer takes a

lot of experience and time. Developing a good smart fuzzer within two weeks is usually infeasible. If you are still interested, please talk to the instructor to receive this bonus challenge!

You may refer to “Directions” for a step-by-step guide of fuzzer development. Your fuzzer will take an initial seed and two arguments, `prng_seed` and `num_of_iterations` as input. Your fuzzer will generate output to stdout, and the output will be used as the input to target programs that will be fuzzed. Your fuzzer may write anything to stderr; any data going to stderr will be discarded. Your fuzzer must behave in a deterministic manner. This means if you execute your fuzzer twice with the same set of parameters, you will get the same mutated output in both executions.

Your code should be understandable and well documented in case the instructor or the TA decides to manually grade your submission.

You will be given a set of three target programs and a seed input file for each of them. You may test your fuzzer against the target programs with different PRNG seeds and numbers of iterations. You may monitor the return value of target programs after running them with input that is generated by your fuzzer. When the target program crashes, write down your `prng_seed` and `num_iterations`.

Hints: Note that your fuzzer will have access to the executable itself. You may build a dictionary of strings in the executable to help your fuzzer generate input with higher quality.

Directions

Here is a step-by-step description of how you will develop the fuzzer. Your fuzzer will take an initial seed and two arguments, `prng_seed` and `num_of_iterations` as input. Your fuzzer will generate *one* output in a deterministic manner, i.e., for the same combination of initial seed, `prng_seed`, and `num_of_iterations`, your fuzzer should generate the same output file. This output file will be used as input to the fuzzing target.

- Your fuzzer will be executed using the following command line:

```
./fuzzer prng_seed num_of_iterations
```

where `prng_seed` is a 32-bit integer that you may use to seed your PRNG(s), and `num_iterations` is a 32-bit integer that determines how many iterations the fuzzer will run.

- Your fuzzer will read an initial seed file called _seed_. The _seed_ file is located under the current working directory.

- Optionally, your fuzzer will use `prng_seed` to seed any PRNG(s) that will be used during fuzzing.

- Your fuzzer will iterate for `num_iterations` times. In every iteration, your fuzzer should change each byte of the input to a random byte with 13% probability. Do not change or overwrite the `seed` file on the disk.

- Your fuzzer will extend the input string by adding 10 random characters to the end of the input every 500 iterations. Again, do not change or overwrite the `seed` file on the disk.

- After all iterations, your fuzzer will write the mutated input to stdout. This means that your fuzzer should not write to stdout anything that is not part of your mutation result. For debugging purposes, your fuzzer may write to stderr. Any data that is written to stderr will be ignored.

- Your fuzzer terminates.

Evaluation

Your fuzzer will be first evaluated against the three test programs with the provided `prng_seed` and `num_iterations`. You will get 10 points for crashing each of them.

Additionally, your fuzzer will be tested on a comprehensive test suite with both programs that are similar to the test targets that you have and programs that are not related to any test targets. There will be ten (10) test programs. For each test program, your fuzzer will have one (1) hour of CPU time. Crashing each target will give you 10 points.

This project accounts for 30% of your final grade of this course.

Submission Directions for Project Deliverables

Your fuzzer code. It should include a `fuzzer` executable that takes command line parameters as previously described.

For each test program, submit a text file named after the test program with the `prng_seed` and `num_iterations` for which your fuzzer generates a crashing input. If the test program is called `test`, then your test file should be named `test.txt`.

A document listing any dependencies that your fuzzer has, and detailing your input generation strategy.