Fuzzing with AFL

Sara Ersson

May 2019

What the Fuzz?

Writing secure programs is not an easy task. Just because no bugs are detected does not ever mean that there or none. Fuzzing is a technique to automatically test software with input to find interesting and unexpected behaviours, and see how well the software withstands possibly malicious input. [1]

In this report a fuzzing tool, named AFL, is going to be used together with different test cases to test 20 different GNU commands.

How Fuzzing Works

Different Fuzzers

There are three different types of fuzzers; generation-based, mutation-based, and a mix of both. Generation-based fuzzers create input from scratch, whilst mutation-based ones generate input by mutating a given *template* input. [2]

Not only are there different types of fuzzers, they can also be either *dumb* or *smart*. A dumb fuzzer is not aware of the input structure so it either generates random test input or mutates given input in a random manner. A smart fuzzer however has learned how the input to the program should look like and either generates, or mutates given input, the way it does for a reason.

In other words, fuzzers can be aware of input structure. They can also be aware of program structure by tracing the target program and generating input from its behaviour, which is why some enable white-box testing and others enable grey- or black-box testing.

AFL

AFL, or American Fuzzing Lop, is a fuzzing tool which is both mutation-based and smart. AFL also comes with a coverage tool for measuring different types of coverage, such as branch and line coverage.

Workflow

Setup

Firstly, a docker image was downloaded from the course page on github, prepared with coreutils and coreutils binaries rebuilt with debugging information. The docker image was started with 'docker run -it –privileged image_id bash'. The privileged flag was added because it was going to be needed to set core dumps when running AFL. The official Git repo of GNU coreutils was cloned in the container and tag v8.25 was checked out.

To enable the afl fuzzing '#include "../../afl-2.52/experimental/argv_fuzzing/argv-fuzz-inl.h"' was added in each command source file. Furthermore, to map stdin to argv 'AFL_INIT_ARGV();' was added in the beginning of the main method in each command source file.

For configuration, permitting running as root and setting the compiler to the afl-compiler './configure FORCE_UNSAFE_CONFIGURE=1 CC="../afl-2.52/afl-gcc"' was run in the coreutils folder. To CFLAGS '-fprofile-arcs -ftest-coverage' was added since it was required for coverage.

The Makefile was edited in in such way that it would not treat warnings as errors anymore, by deleting the Werror flag. Finally 'make clean all' was run. The building of the project had to be interrupted manually when all coreutils binaries had been created since the Makefile starts building the manual pages in the end.'

Creating Input and Running AFL

The AFL pipeline works by giving the fuzzer an input directory, an output directory and a binary to run. An example is running 'afl-fuzz -i input_cat -o output_cat ./cat' for the command 'cat'.

There is one input directory and one output directory for each command to be run, and each input directory contains several test files. Each test file consists of an input to the binary. Examples of different inputs are shown below:
-A
-B
--version
--help
-s
-t
-n
-v
-g -t

Data regarding the fuzzing, such as fuzzed input and crashes etc., is stored in the corresponding output directory. The fuzzer was run at least one cycle per binary and then stopped manually, and the coverage analysis could be viewed by running 'gcov cat', for the cat command. In the beginning of the analysis line coverage in percentages could be viewed.

Minimizing and Formatting the Output

To reduce redundancy afl-cmin was run, which minimizes the amount of test cases, but keeps the coverage. The result was then stored in the corresponding output directory:
afl-cmin -i queue -o min_result ../cat

One new file per binary was then created and filled with one test input per line.
cat min_result/* > ../../results/cat.txt

Finally, to make the test inputs utf-8-friendly below script was run, which replaces all non-utf-8 characters with '@':

cat cat.txt —tr -c '[:print:][:cntrl:]' '@' > final_cat.txt

Take-Home Message

For everyone wanting to try fuzzing I would give following advice:
"Find a fuzzer with which is well-documented so you do not have to struggle so much with the setup. Be patient and let the fuzzer run for quite a while, since the coverage can become higher even if you do not think it will. With a proper fuzzer and patience you could receive great help with your work.

References

[1] A. Helin. (2019, Feb.) A crash course to radamsa. [Online]. Available: https://gitlab.com/akihe/radamsa

[2] M. Hillman. (2013, Aug.) 15 minute guide to fuzzing. [Online]. Available: https://www.mwrinfosecurity.com/our-thinking/15-minute-guide-to-fuzzing/