Fuzzing Lab: Learn About Fuzzing

Assigned: 20191119

Due: 20191210 23:59

Qian Dan(18212010023@fudan.edu.cn) and Xu Chaozhi(18212010044@fudan.edu.cn) are the lead person for this assignment.

1 Introduction

The purpose of this lab is to learn about fuzzing which is a powerful technique in vulnerability exploiting. In order to be familiar with the basic conception of fuzzing and experience on real world fuzzing tools, you'll need to complete two sections. In **Section A**, you'll learn to make use of **AFL**, a state-of-art fuzzing tool, to fuzz on a given target. In **Section B**, you'll need to write your own demo fuzzer, and try to trigger crashes on the given simple target.

2 Missions

2.1 Section A : AFL

Instructions

The source code of afl-2.52b is given under this directory, or you can download by yourself from the official website.You'll have to follow these steps to complete this section:

1. **Install AFL**. It is easy to install AFL, you only have to compile the source code by `make` which will generate executable files under the current directory. (If you like , you can use `make install` to install AFL into your system, but it is not necessary.)
2. **Read Documents**. To learn about how to use AFL, you need to read the documents under `afl-2.52b/docs` .
3. ***(optional)*** Patch AFL with **laf-llvm-pass**. It is costly to run origin AFL on the target program, almost a whole day is needed to trigger one crash. We suggest to patch with **laf-intel** which will significantly enhance the performance of AFL. The source and usage can be found on the laf-intel website

4. **Compile the target program**. The source code of target program is given under this directory, compile with `afl-gcc` or `afl-clang-fast` and you'll find the target binary under `src`

5. **Run AFL on the target**. Follow the instructions in the documents to run AFL on the target program, you'll get one crash after almost a whole day without patched in **Step 3**, while just a few minutes with patched.

Technique Notes

1. If you are going to try **llvm-mode** or patch AFL with **laf-llvm-pass**, you need to follow thses steps:

   a) Install **clang**. On ubuntu, `sudo apt install clang-3.8` . If `clang-3.8` does not work well, you could try `clang-3.6`

   b) Compile AFL with **clang**. Set `CC=clang-3.8 CXX=clang++-3.8` before compile. then `make` .

   c) Follow the Instructions of **llvm-mode** or **laf-intel** to do the rest.

2. When compile the target program, you shuold have the dependency lib `acl` installed by `sudo apt install libacl1`, then `./configure CC=AFL_PATH/afl-gcc LIBS="-lacl"`. If you installed llvm-mode, you can run `./configure CC=AFL_PATH/afl-clang-fast LIBS="-lacl"`

3. Sometimes it is hard to find the **executable files** after compile, you could set output directory like this `./configure –prefix=your_install_directory`

Handin Instructions(sectionA)

You need to pack **the whole output directory** in your AFL command "`afl-fuzz -i input -o output md5sum -c @@`" into **afl-output.tar**.

2.2 Section B : simple fuzzer

Instructions

In this section you'll write a simple fuzzer to fuzz on the ***coreutils ls*** program which injected with vulnerabilities. You need to find out the inputs crashing the target program with your own fuzzer. To complete your fuzzer, there are at least three parts you need to implement:

1. **Runing the target program**. In ***C++*** you could use `popen` , which will run the command you give & receive the output message
2. **Inputs mutation**. You need to mutate the input to generate a different one as the input of the target program each round. There are many mutation strategies you could use, and you need to find out yourself by searching on the internet. I suggest to use more than two strategies for a better performance, and you could select one or more strategies randomly to mutate the input each round. The following gives 6 simple mutation strategies:
   a) write random uint8 data to random position.
   b) bit flip in random position.
   c) increase the uint8 data by one in random position.
   d) decrease the uint8 data by one in random position.
   e) insert random uint8 data to random posstion.
   f) remove random uint8 data in random position.
3. **Output processing**. There two or more ways to get to know if the target program normally returned or crashed, depending on the method you take to run the program. You could analyze the output message or the return value. There is a sample input string causing crash under `targets/sectionB/readme.txt`
4. **Inputs saving**. You need to save the inputs leading to crash generated by your fuzzer
5. **Information display**. It is recommended to print proper information along your fuzzing process

The target program is under `targets/sectionB/` , and it is a executable file. Enter the the directory `SectionB` and run the command `./ls known_crash_input` to check the crash output, or `./ls normal_input` to the see the normal output. The file `targets/sectionB/tips.txt` is useful for you to complete this section.

Technique Notes

You can use ***C/C++***, ***JAVA***, ***PYTHON*** to complete this section

Handin Instructions

1. The source code of your program.
2. The excutable file of your program.
3. The document of how to run your program, what dependencies needed.
4. The result of your fuzzer, especially the inputs causing crashes.

Handin File List(sectionB)

- README.pdf
- excutable file
- source code
- crash-inputs

3 Handout File List

- FuzzingLab.pdf
- afl-2.52b
- targets
  - sectionA
    md5sum
  - sectionB
    ls
    readme.txt
    tips.txt
    runFuzzer.py
    lab4_fuzzer.pptx

4 Evaluation

Section A (40 points)
One unique crash for 8 points, which means you can get the whole 40 points if you fuzzed out at least 5 unique crashes.

Section B (60 points)
1.Implement basic 4 parts of a fuzzer in your source code. (10 points)
2.The fuzzer can normally run. (10 points)
3.The performance of your fuzzer. (15 points)
4.The fuzzer can find out crashes on the evaluation program. (25 points)

Note

Do Not Cheat or Fake.