Fuzzing Lab

Assigned: 20201201

Due: 20201221 23:59

---

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 Tasks

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.
2. **Read Documents**. To learn about how to use AFL, you need to read the documents under `afl-2.52b/docs`.
3. **Compile the target program**. The source code of target program **uniq** is given under this directory, compile with `afl-gcc` or `afl-clang-fast` and you'll find the target binary `uniq` under `src`.
4. **Run AFL on the target**.

Handin Instructions

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

2.2 Section B : simple fuzzer

Instructions

In this section you'll write a simple fuzzer to fuzz on the base64_encode 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:
    1. write random uint8 data to random position.
    2. bit flip in random position.
    3. increase the uint8 data by one in random position.
    4. decrease the uint8 data by one in random position.
    5. insert random uint8 data to random posstion.
    6. 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 analyse the output message or the return value. There is a sample input file causing crash under `targets/sectionB/`.
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 which input is file path. Enter the the directory `Section B` and run the command `./base64_encode crash_sample_input` to check the crash output, or `./base64_encode normal_sample_input` to the see the normal output.

Technique Notes

1. 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

- README.pdf
- excutable file
- source code

    - You should provide comments in critical parts
- crash-inputs

Section A (10 points)

1. Considering the performance of virtual machine and laptop, please submit the screenshot which can represent you have tested AFL and LAVA-M successfully. Besides, please highlight the time in screenshots.

Section B (90 points)

1. Implement basic 4 parts of a fuzzer in your source code. (**25 points**)
2. The fuzzer can normally run. (**15 points**)
3. The performance of your fuzzer. (**20 points**)
4. The fuzzer can find out crashes on the evaluation program. (**30 points**)

Note

Do Not Cheat or Fake.