CS-52700-LE1 – Spring 2020 – Homework Assignment 5

This homework assignment is about tools for automated binary analysis and bug finding.
This homework assignment is due on:
April 26th, 2020, at 11 PM EDT.

Please, ask general questions about the homework on Piazza, so that everyone can benefit from the answers. However, do not include solutions (or part of solutions) in your public questions.

General Requirements

Submit your homework as a *single* `.tgz` file (tar gzipped file). Create your `.tgz` file using the following command:
`tar czf ⟨yourfile.tgz⟩ ⟨yourfolder⟩`
The name of your submission must be `⟨your_PurdueID⟩.tgz`
Substitute `⟨your_PurdueID⟩` with your actual PurdueID (a PurdueID has a format similar to `antob12`).

When unpacked, the `tgz` file must have the following directory structure:

```
⟨your_PurdueID⟩
    task01
        README.txt
        input1_t1
        asan_t1_asan
        asan_t1_output
        ⟨any other code/script you may have used to solve the task⟩
    task02
        README.txt
        input1_t2
        input2_t2
        ⟨any other code/script you may have used to solve the task⟩
    task03
        README.txt
        dict1
        input1_t3
        input1minimized_t3
        ⟨any other code/script you may have used to solve the task⟩
    task04
        README.txt
        input1_t4
        postprocessing.so.c
        postprocessing.so
        ⟨any other code/script you may have used to solve the task⟩
    task05
        README.txt
        angr_t5.py
        input1_t5
        ⟨any other code/script you may have used to solve the task⟩
    task06
        README.txt
        angr_t6.py
        input1_t6
        ⟨any other code/script you may have used to solve the task⟩
    task07
        README.txt
        angr_t7.py
        input1_t7
        ⟨any other code/script you may have used to solve the task⟩
```

The `README.txt` files must contain a short description (one or two paragraphs) about how each task has been solved or attempted to be solved. The `README.txt` must be ASCII text files.

I will grade this homework by assigning the same amount of points to each task. I will remove from your grade about this homework your lowest-grading task. This implies that if you score perfectly in 6 tasks and you skip one task, you will still get a 100% grade.

All the tasks assume that you will use a Ubuntu 18.04 64bit machine, and they will be graded using a machine of this type.

You are free to use a virtual machine. You will need to install packages on the machine. Therefore, you should have root access on it. Task 7, may require to have a significant amount of RAM available (about 5GB). We will not provide virtual machines for this assignment. If you really cannot satisfy the requirements mentioned above, contact the TA as soon as possible, and we will work together to find a solution.

In many tasks, I will ask you something like:
"In the file named `xyz`, provide an input that triggers the condition `abc`"
This means that you will need to provide a file named `xyz`. I will test if the condition `abc` is triggered when running:
`cat xyz | ./program`
When I mention "crashing" a program, it means triggering a "Segmentation Fault" exception.

For some tasks, you will need to use AFL, version 2.52b, with QEMU support. You are free to download and install AFL with QEMU support by yourself. However, I provide it to you here:
`https://www.cs.purdue.edu/homes/antoniob/shared/afl-2.52b_qemu.tar.gz`
All the solutions of tasks requiring AFL will be tested using as initial input file the file `aflinput`.

Some tasks require using the tool `angr`. To install it, these are the required steps:

```
sudo apt install python3
sudo apt install python3-dev
sudo apt install python-pip
sudo apt install virtualenvwrapper
```

At this point open a new shell (or reboot). Then run:

```
mkvirtualenv -p `which python3` angr
```

At this point you are in the angr Python virtual environment, if you want to enter into it again use the command: `activate angr`

To install `angr`, while the `angr` virtual environment is activated, run:
`pip install ipython`
`pip install angr`
At this point, by running `ipython`, IPython should start showing "Python 3" in its initial prompt. In IPython, executing `import angr` should not give any error.

In Task 5, Task 6, and Task 7, the binaries you have to analyze and the Python code you have to write is similar to what shown here:
`https://github.com/angr/angr-doc/tree/master/examples/CADET_00001`
You should read carefully the provided solution, experiment with it, and use it as a starting point for solving these three tasks.

Task 1

The provided `asan_t1.c` file contains the source code of a simple program. The `asan_t1` file contains its compiled version.

1. In the file named `input1_t1`, provide an input exploiting `asan_t1` so that it prints "Congratulations you are now an admin!".

2. Compile the code of `asan_t1.c`, enabling Address Sanitizer, in a file named `asan_t1_asan`. You must use `clang-9`, which is not installed by default in Ubuntu 18.04. You can install it by running:
   `sudo apt-get install clang-9`
   To compile the code using Address Sanitizer use the following command:
   `clang-9 -O0 -fsanitize=address -fno-omit-frame-pointer ...`

3. Verify that the input in `input1_t1` does not exploit `asan_t1_asan`, since the exploited memory corruption bug is detected by Address Sanitizer. Save the output of Address Sanitizer when running `cat input1_t1 | ./asan_t1_asan` in a file named `asan_t1_output`.

Task 2

The provided `asan_t2` binary is an easy program. The provided `asan_t2_asan` binary is the same program compiled used Address Sanitizer.

1. In the file named `input1_t2`, provide an input exploiting `asan_t2` so that it prints "Congratulations! You got the secret number!".

2. In the file named `input2_t2`, provide an input exploiting `asan_t2_asan` so that it prints "Congratulations! You got the secret number!".

Task 3

Use AFL to find a crashing input for the provided program `fuzz3`. Specifically:

1. The provided program requires you to insert specific strings. AFL typically cannot easily fuzz programs like this, since it struggles in randomly finding correct strings. However, you can provide to AFL a dictionary file, containing "tokens" that AFL will use during fuzzing. By reversing the `fuzz3` binary, create a dictionary file, suitable to be used with AFL, and save it in a file named `dict1`. I will test your dictionary file by using it with AFL to fuzz the `fuzz3` binary for about 5 minutes. A proper dictionary file should allow finding a crash in about one minute.

2. Use AFL to find a crashing input for the `fuzz3` binary. Save it in a file named `input1_t3`. You will need to use AFL with QEMU support and the previously created dictionary file. Verify that the `input1_t3` input crashes `fuzz3`.

3. Minimize, using the `afl-tmin` utility, the provided `input1_t3` file, and save it in a file named `input1minimized_t3`. Verify that the `input1minimized_t3` input crashes `fuzz3`.

Task 4

Use AFL to find crashing inputs for the provided program `fuzz4`. `fuzz4` is similar to `fuzz3`, but it contains an initial CRC check. Specifically:

1. The provided program requires you to start any input with a CRC (a value, stored at the beginning of the input, used to check the integrity of the rest of the input). AFL typically cannot easily fuzz programs like this, since it is extremely unlikely for a randomly generated input to have a correct CRC. However, it is possible to provide a postprocessing library to modify every input generated by AFL so that it contains a proper CRC. You can find more information here:
   `https://github.com/google/AFL/blob/master/experimental/post_library/post_library.so.c`

   By reversing the `fuzz4` program, you can understand how the CRC is computed and create a proper postprocessing `.so` file. Save your postprocessing `.so` file in a file named `postprocessing.so` and its source in a file named `postprocessing.so.c`. I will test your postprocessing file by using it with AFL to fuzz the `fuzz4` program, for about 5 minutes. A proper postprocessing file will allow to find a crash in less than a minute.

2. Use AFL, together with the created postprocessing library, to create a crashing input for the `fuzz4` program. Save it in a file named `input1_t4`. Verify that the `input1_t4` input crashes `fuzz4`.

Task 5

Write a Python script using `angr` (named `angr_t5.py`) that generates a file named `input1_t5`. This file must contain an input that, when provided to the binary `angr1`, makes it printing the string "EASTER EGG!". Your script must use angr's symbolic execution to automatically find and input reaching the basic block where the string 'EASTER EGG!" is printed.

Task 6

Write a Python script using `angr` (named `angr_t6.py`) that generates a file named `input1_t6`. This file must contain an input that, when provided to the binary `angr2`, makes it printing the string "EASTER EGG!". Your script must use angr's symbolic execution to automatically find and input reaching the basic block where the string 'EASTER EGG!" is printed. For this task, you should create a `symprocedure`[1] to summarize the function named `mm` in `angr2`, since `angr` cannot properly symbolically execute it otherwise.

Task 7

Write a Python script using `angr` (named `angr_t7.py`) that generates a file named `input1_t7`. This file must contain an input that, when provided to the binary `angr1`, makes it crash. Your script must use angr's symbolic execution to reach a state in which the instruction pointer is unconstrained (which implies that it is controllable by user's input). This means that concretizing the content of standard input will most likely lead to a input which brings the execution to an non-executable memory location (thus, the crash).

[1] Read: https://github.com/angr/angr-doc/blob/master/docs/simprocedures.md