CS-52700 - Spring 2024 - Homework Assignment 5

This homework assignment is about address sanitizer, fuzzing with AFL, and SECCOMP. This homework assignment is due on:

April 5th 2024, at 11:59PM 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 -czvf <your_purdue_id>_HW5.tgz`

Substitute **<your_purdue_id>** with your actual PurdueID (a PurdueID has a format similar to `antob12`, and it is all lowercase).

When unpacked, the tgz file must have the following directory structure **EXACTLY**.

```
<your_purdue_id>_HW5/
  task01/
    README.txt
    input1_t1
    input2_t1 (optionally, as explained in the description of Task 1 below)
    asan_t1_asan
    asan_t1_output
    <supplementary files referenced in README.txt as needed>
  task02/
    README.txt
    dict1
    input1_t3
    input1minimized_t3
    <supplementary files referenced in README.txt as needed>
  task03/
    README.txt
    input1_t4
    postprocessing.so.c
    postprocessing.so
    <supplementary files referenced in README.txt as needed>
  task04/ (only for students in the in-person section)
    README.txt
```

```
    flag.txt
    <supplementary files referenced in README.txt as needed>
  task05/ (only for students of the distance learning section)
    README.txt
    flag.txt
    <supplementary files referenced in README.txt as needed>
```

The **README.txt** files must contain short descriptions (one or two paragraphs) describing how each task has been solved. Be succinct in your description, if you are in doubt do ask the TA. **As supplementary files, you are expected to include any script that you used to solve the task.**

You can find the files mentioned in this README in the hw5_527_Spring2024_files.tgz file on Brightspace.

When I mention "crashing" a program, it means triggering a "Segmentation Fault" exception. When I mention a "crashing" input, it means a file passed as an input to the program that causes a " Segmentation Fault" exception when used like this:

```
cat <input_file> | ./program
```

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

For fuzzing tasks I will test your solution for, at most, 5 minutes. A correct solution will cause AFL to find a crashing input in about 1 minute or less. All the solutions of the tasks requiring AFL will be tested using as initial input file the file: `aflinput`.

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`. To compile code using Address Sanitizer use the following command: `clang -O0 -fsanitize=address -fno-omit-frame-pointer ...`
3. Verify whether or not the input in `input1_t1` exploits `asan_t1_asan.` The most likely scenario is that the memory corruption triggered by input1_t1 is detected by Address Sanitizer, therefore `input1_t1` does not exploit `asan_t1_asan`.
   If this **is** the case, save the output of Address Sanitizer when running: `cat input1_t1 | ./asan_t1_asan` in a file named `asan_t1_output`.

If this **is not** the case, please provide an additional input file in your task01 folder, named `input2_t1`. This additional input needs to trigger memory corruption in `asan_t1_asan`, in a way in which it is detected by Address Sanitizer. Save the output of Address Sanitizer when running:
`cat input2_t1 | ./asan_t1_asan` in a file named `asan_t1_output`.

Task 2

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 an appropriate 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 3

For this task, you do not have to specify a dictionary file. I will not use any dictionary file when testing your solution.
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.

   By reversing the `fuzz4` binary, 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 finding 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`.

Connecting to the machines (for Task4 and Task5)

To solve your tasks, you need to connect to your assigned virtual machine. To do so, you can use the command:
`ssh -p <your_assigned_port> <Purdue_id>@<IP>`

To know the IP address of your assigned virtual machine and your assigned port, please visit the following page:
https://www.cs.purdue.edu/homes/song464/cs527/hw5

We may need to modify these IP addresses. Therefore, in case you are suddenly unable to connect, you should check that page to verify if your assigned IP changed. We may also need to reset the machine and delete all the files stored there. For this reason, always keep local copies of any file stored in your assigned machine.

Once you login into your assigned machine, you will see some instructions about how to switch to the user associated with each task.

In general, your goal is to read the content of the file **`flag.txt`** in each task folder. Given how permissions are set, you cannot directly read this file. However, by reverse engineering and/or exploitation of the setuid binary contained in each task folder, you can manage to read the content of the flag file.

Task 4:

For task4, the flag is stored "flag" instead of "flag.txt".

Task 5 hint:

You may want to check this: