CS-52700 - Spring 2024 - Homework Assignment 6

This homework assignment is about using Rust, `angr`, and `angr`'s symbolic execution capabilities, for automated binary analysis and bug finding. This homework assignment is due on: **April 21st 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>_HW6.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>_HW6/
  task01/
    README.txt
    angr_t1.py
    input1_t1
    <supplementary files referenced in README.txt as needed>
  task02/
    README.txt
    angr_t2.py
    input1_t2
    <supplementary files referenced in README.txt as needed>
  task03/
    README.txt
    angr_t3.py
    input1_t3
    <supplementary files referenced in README.txt as needed>
  task04/
    README.txt
    calc.rs
```

The **README.txt** files must contain short descriptions (one or two paragraphs) describing how each task has been solved. If the flag has not been found then please explain

what have you done far and your thought process as to what's needed. Be succinct in your description, if you are in doubt do ask the TA.

All the tasks assume that you will use a Ubuntu 20.04 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.

In some tasks, I will ask you something like:
"In the file named <xyz>, provide an input that triggers the condition <abc>"
This sentence 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.

These tasks require using the tool `angr`. To install it, follow the instructions provided by during the lectures and the labs. Please do this early, so that we can have enough time to troubleshoot possible issues.

In these tasks, the binaries you have to analyze and the Python code you have to write is similar to what shown at: https://github.com/angr/angr-doc/tree/master/examples/CADET_00001
The source code used to generate the `angr1` and `angr2` binaries is a modified version of this code:
https://github.com/CyberGrandChallenge/samples/blob/master/examples/CADET_00001/src/service.c

You should read carefully the solution provided in the `angr` documentation, experiment with it, and use it as a starting point for solving these three tasks.

Task 1

Write a Python script using `angr` (named `angr_t1.py`) that generates a file named `input1_t1`. This file must contain an input that, when provided to the binary `angr1`, makes it prints the string "*EASTER EGG!"*. Your script must use `angr`'s symbolic execution to automatically find an input reaching the basic block where the string "EASTER EGG!" is printed.

Task 2

Write a Python script using `angr` (named `angr_t2.py`) that generates a file named `input1_t2`. This file must contain an input that, when provided to the binary `angr2`, makes it prints 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 must use a simprocedure (read

https://github.com/angr/angr-doc/blob/master/docs/simprocedures.md) to summarize the function named `mm` in `angr2`, since `angr` cannot properly symbolically execute it otherwise.

Task 3

Write a Python script using angr (named `angr_t3.py`) that generates a file named `input1_t3`. 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, in this case, implies that it is controllable by user's input). This means that concretizing the content of standard input will most likely lead to an input bringing the instruction pointer to a non-executable memory location (thus, the crash).
This task may require to have a significant amount of RAM available (about 8GB). 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. **To keep the memory consumption low, you should discard deadended paths.** To do this automatically, you can generate a `SimulationManager` using the `auto_drop` option in the following way:

```
sm = project.factory.simulation_manager(auto_drop=["deadended"])
```

Task 4

Create, writing it in Rust, a simple calculator that is a console program, accepting user input from standard input and writing to standard output. Your calculator must be implemented as explained in the following specification, but you are free to implement as you prefer anything not strictly specified.

Your calculator must perform 5 basic arithmetic operations (addition, subtraction, multiplication, integer division, and real division), specified, respectively, by the characters +, -, *, //, /. Additionally, if the string "exit" is specified as operation, the program must terminate.

Each one of these 5 operations must be implemented in a separate function. Each function should get, as input, 2 integers of type u32.

Integer division should be used when the user inserts // as the operation, while real division should be used when the user inserts / as the operation. Assuming a=5 and b=2, the result of the integer division (a//b) is supposed to be 2 (i.e., fractional part is discarded), while the result of the real division (a/b) is supposed to be 2.5. If needed, you can use the expression: `(variable as f32)`
to convert an integer variable to a floating point variable.

In addition, your calculator should accept an optional command line argument. If, and only if, the user provides a command line argument, when terminating, your calculator should save in the filename specified by this command line argument the content of the result of the last performed operation. The result should be saved as an ASCII string and terminated by a new line.

Save your code in a single file written in Rust and name it: calc.rs

The output of your program should be formatted exactly according to the following examples:

Example 1:

```
$ ./calc
Input operation: +
Number 1: 3
Number 2: 4
Requested operation was addition (+) and the result is: 7
Input operation: -
Number 1: 5
Number 2: 2
Requested operation was subtraction (-) and the result is: 3
Input operation: exit
```

[At this point the program must terminate and no file is saved since your calculator was called without any command line argument]

Example 2:

```
$ ./calc savedfile
Input operation: +
Number 1: 3
Number 2: 4
Requested operation was addition (+) and the result is: 7
Input operation: -
Number 1: 5
Number 2: 2
Requested operation was subtraction (-) and the result is: 3
Input operation: exit
```

[At this point the program must terminate and it must create a file named "savedfile". The file named "savedfile" must contain exclusively the string "3" followed by a new line]

Example 3:

```
$ ./calc
Input operation: /
Number 1: 5
Number 2: 2
Requested operation was real division (/) and the result is: 2.5
Input operation: //
Number 1: 5
Number 2: 2
Requested operation was integer division (//) and the result is: 2
Input operation: exit
```

[At this point the program must terminate and no file is saved since your calculator was called without any command line argument]