CSE 5472: Symbolic Execution Lab

Objective

Learn about the merits and challenges of using binary symbolic analysis to analyze program behaviors, including to discover vulnerabilities.

Deliverable

1. Task 1 solution code
2. Task 2 solution code

Environment

Please use a Linux environment (Debian or Ubuntu preferred). Virtual machines are fine.

If you plan to use `stdlinux`, you will need to setup a virtual environment in order to install `angr`.

Provided Materials

- `fauxware`: Target program for Task 1.
- `task1.py`: Template Python script for creating the solution to Task 1.
- `goose_publisher_example`: Target program binary for Task 2.
- `task2.py`: Template Python script for creating the solution to Task 2.

Recommended Tools

- angr

Tasks

1. Complete the script `task1.py` so that it prints out the correct password for `fauxware`.
2. Complete the script `task2.py` to symbolically analyze `goose_publisher_example` and print all functions that use `strcpy`, which is a notoriously unsafe libc function. Determine which functions that call `strcpy` are potentially vulnerable and print them separately.

Grading

- Task 1 Solution Code:
  - Does `task1.py` print the correct password for `fauxware`? **10 points**
- Task 2 Solution Code:
  - Does `task2.py` build a control flow graph of `goose_publisher_example`? **5 points**
  - Does `task2.py` identify the address of `strcpy`, either the `SimProcedure` or the PLT stub, using the CFG? **10 points**
  - Does `task2.py` print all the functions in `goose_publisher_example` that invoke `strcpy` based on the CFG? **15 points**
  - Does `task2.py` print which of these functions are *potentially vulnerable*? See Hints for details. **10 points**
- **Total Possible Points: 50**

Notes

- The TA will use *their* copy of `fauxware` and `goose_publisher_example` to run your scripts. Therefore, avoid using hard coded addresses that may change!

- For Task 2, please print your answers at the end of `task2.py`'s output. angr's logging is verbose and interspersing your answers between angr messages will make the TA's job harder and possibly lead to grading mistakes.
- For the latter parts of Task 2, do not think that the point values indicate the number of functions in the correct answer. We will use the formula: `TP / (TP + FN + FP)`.

References

- angr usage documentation
- angr API documentation

Hints

What is a potentially vulnerable function?

For the purposes of this lab, we will define a potentially vulnerable function as one that can reach any executable address starting from a basic block that calls `strcpy`. In other words, the instruction pointer (a.k.a. program counter) becomes symbolic and has no constraints.

To test this in angr, we recommend making use of `project.factory.call_state` and a `SimulationManager`. See the angr API documentation for more details.

Finding Addresses by Symbol Name

The target binaries in both tasks are not stripped, meaning that if you have an angr project `p`, and you want to find the address of function `foobar`, you can do so using its symbol:

```
addr = p.loader.find_symbol('foobar').rebased_addr
```

Task 1 Miscellaneous

- Use `auto_load_libs=False` when creating the angr project to make the analysis faster.
- When the correct password is entered into `fauxware`, the program enters a function named `accepted`.
- The correct password is alpha-numeric, i.e., `[0-9A-Za-z]+`.

Task 2 Miscellaneous

- Use `auto_load_libs=False` and `CFGFast` when creating the angr project and CFG to make the analysis faster.