Security Through The Software Lifecycle: Fuzzing Project (2023-2024)

Version 1.0

Contents

1 Introduction

In this project you will apply grey-box fuzzing and symbolic execution to discover a set of memory corruption vulnerabilities. You will use of two state-of-the-art tools, namely *AFL++* and *KLEE*.

1.1 *AFL++*

AFL++ is an improved version of the original *American Fuzzy Lop* fuzzer. It is arguably one of the most well-known fuzzers and it has an extensive documentation. When using *AFL++*, a modified compiler is used to compile the program to be fuzzed. This compiler will add instrumentation to the program which counts branch coverage (see our lecture). By default *AFL++* only detects bugs that cause the program to crash, but by enabling (address) *sanitizers* during the compilation process it can also detect other kinds of bugs.

A *quick start* introduction to *AFL++* can be found on `https://github.com/AFLplusplus/AFLplusplus/blob/stable/README.md`. More in-depth information on how to fuzz programs when the source code is available can be found at `https://aflplus.plus/docs/fuzzing_in_depth/`. Read both links before starting the project. To get a full grade on the project, you need to have demonstrated that you understand most of the points touched upon in the second link.

For more in-depth information on the core aspects of *AFL++*, you can read the technical whitepaper of its predecessor AFL: `https://github.com/mirrorer/afl/blob/master/docs/technical_details.txt` .

1.2 KLEE

KLEE, like it's predecessor EXE, is a dynamic symbolic execution engine build on top of the LLVM compiler infrastructure. KLEE operates on the Intermediate Representation (IR) used in the LLVM compiler, hence to run a program in KLEE, source code first needs to be compiled to LLVM IR. KLEE can be used to detect a wide range of bugs such as memory corruption errors, assertion errors and integer overflows or to generate a high-coverage *test suite*.

KLEE is cited in 250+ papers that make use or extend the tool, making it one the most popular symbolic execution engines to date.

To get familiar with KLEE, first go through the basic KLEE tutorial: `https://klee.github.io/tutorials/testing-function/`. This website also contains documentation and other tutorials on how to use KLEE. As a second step to get familiar with KLEE, you must read the tutorial on using a symbolic environment: `https://klee.github.io/tutorials/using-symbolic/` Finally, read how KLEE can be used to test bigger projects such as the coreutils tools, which will learn you how to compile bigger programs with LLVM so they can be tested using KLEE: `https://klee.github.io/tutorials/testing-coreutils/`.

For more in-depth information on how KLEE works, you can optionally read its paper `https://www.doc.ic.ac.uk/~cristic/papers/klee-osdi-08.pdf`

.

1.3 Execution environment

Both AFL's and KLEE's maintainers provide an up-to-date Docker image with all related tooling preinstalled. We tested all exercises using these images and we recommend you to also use these when solving the project. When using Ubuntu 22.04, you can install Docker Engine using the following command:

```
sudo apt install docker.io
```

You can open a shell running inside AFL's and KLEE's respective containers using one of the following commands:

```
sudo docker run --rm -ti aflplusplus/aflplusplus
sudo docker run --rm -ti --ulimit='stack=-1:-1' klee/klee
```

Adding the following argument to the *docker run* command will mount the *current working directory* inside the Docker container at /mnt-host.

```
--mount type=bind,source=.,target=/mnt-host
```

For more details on using KLEE with Docker, see `https://klee.github.io/docker/`. Note that KLEE will create a new output directory `klee-out-X` every time you run it. To save disk space, you can periodically remove all these output directories.

1.3.1 Apple ARM64 architecture

You can also run the above Docker images on MacBooks that use the ARM64 architecture. You first have to install Docker with Rosetta support. See `https://docs.docker.com/desktop/install/mac-install/` and then click on the tab "Mac with Apple Silicon" for the installation instructions.

When starting a docker instance, add the parameter `--platform linux/amd64` to the docker run command. For instance, you can execute:

```
docker run --rm -ti --platform linux/amd64 aflplusplus/aflplusplus
```

1.4 Background knowledge

This project assumes you are familiar with the C programming language. To refresh your C knowledge, you can read the following guide: `https://beej.us/guide/bgc/html/split/index.html`

2 Challenge 1: Logmein

The ***logmein*** tool reads a password from stdin, computes a hash of the password using a custom hashing algorithm, and spawns a shell if the computed hash matches a hard-coded value. This tool can be used as a *setuid* program to

allow the user to obtain elevated privileges upon entering the correct password. Unfortunately, the program contains a bug, and your objective is to find it.

2.1 Getting started

Compile *logmein.c* using your favorite C compiler. Try to to run it with some input.

2.2 AFL++

Fuzz the program with AFL++. Since *logmein* takes all inputs from *stdin*, creating a harness isn't required. From a high-level you should take the following steps:

1. Compile *logmein.c* with instrumentation for AFL and a method to make memory corruption errors observable. To make this straightforward, AFL ships with a set of scripts which can be used as drop-in replacement for common C and C++ compilers. We recommend to use afl−clang−fast.
2. Manually create a set of meaningful *seeds*.
3. Run AFL with the correct arguments.

AFL should report some *inputs* that would trigger an error. Check for yourself whether these inputs really cause a memory corruption error.

2.3 KLEE

The second part of this challenge consists of finding a memory error in *logmein* using symbolic execution. KLEE has built-in support to return fresh symbolic data when a program attempts to read from *stdin*, for more info check the *using the symbolic environment* section in the KLEE tutorial: `https://klee.github.io/tutorials/using-symbolic/`

HINT: *logmein* makes use of the C standard library, e.g., for the read() function. It is advised to use the standard library that ships with KLEE. In order to do so add −−posix−runtime −−libc=uclibc as a command-line argument to KLEE. Symbolically executing *logmein* roughly consists of the following steps:

1. KLEE operates on LLVM IR, hence *logmein.c* should be compiled to this bytecode in order to be run by KLEE. Check the KLEE tutorial for an example: `https://klee.github.io/tutorials/`
   HINT: Compile the program with *debug symbols* in order to have KLEE report at which location in the program's code an error occurs.
2. Run KLEE with the correct arguments. Make sure to configure KLEE to introduce symbolic data on *stdin*.

KLEE will start symbolically executing *logmein* and generate test files containing program inputs for every distinct path through the program. At least one memory error should be reported. You can find the concrete inputs that would cause these errors using the ktest−tool (`https://klee.github.io/docs/tools/`) program that ships with KLEE. You can use kcachegrind (`https://kcachegrind.github.io/html/Home.html`) to get a graphical overview of the symbolic execution process (run kcachegrind klee−last/run.istats).[1] Optionally, for further details, you can add logmein.c to the source folder path in kcachegrind, see *settings → configure kcachegrind → annotations*.
KLEE adds a *wrapper* around the program's main function in order to implement functionality related to the symbolic environment, such as making program arguments symbolic. The original *main()* function can be found as *__klee_posix_wrapped_main()* in kcachegrind.

3 Challenge 2: Fuzzing a JSON parser

YAJL is a C library providing a **JSON parser**. Your objective is to fuzz the **provided** version of YAJL. We included a vulnerability that can be discovered by using *AFL++* when using common fuzzing settings and appropriate seed inputs. You can find some documentation on how to use this library on it's website: `https://lloyd.github.io/yajl/` . The directory challenge−2−json−parser in the ZIP file contains the vulnerable copy of the YAJL library. In your solution, try to use *AFL++*'s *persistent* mode to speed up the fuzzing process.

Since the code to fuzz itself is a *library*, you are required to create your own *harness*. To create a harness, you can either start from scratch and call YAJL library functions to parse a JSON file, or you can start from example programs that use YAJL and minimize and modify those examples into a harness.

Hint: Have a look at the *dictionaries* directory in the *AFL++* Github repo: `https://github.com/AFLplusplus/AFLplusplus/tree/stable/dictionaries`
Also assure you are using enough seeds as initial inputs to *AFL++*.

4 Challenge 3: Symbolically executing a DNS server

For the third challenge you'll use symbolic execution in order to find a real vulnerability in a **DNS server** that's publicly available on GitHub. For this project, you can use the simplified version that was provided by us. Since KLEE can not handle network I/O at the time of writing, you are required to create a harness to introduce symbolic bytes at the correct program location, and give these symbolic bytes as input to the internal function that you want to test.

[1] On MacOS you can use qcachegrind instead, see example installation instructions on `https://ports.macports.org/port/qcachegrind/`

Hints

1. The **wllvm** tool (`https://github.com/travitch/whole-program-llvm`) will come in handy to compile the program to LLVM IR.

2. Use ***kcachegrind*** to get feedback on the symbolic execution process. Check which parts of the code cause the process to *slow down*, for example, by introducing a lot of symbolic states (forks) or spending a disproportionate amount of time solving path constraints. You will find that a certain coding construct that is often used during debugging significantly impedes the symbolic execution process. Find a way to circumvent this, e.g., by preventing forking at this location.

3. After you have created your harness, it should be possible to run the program under KLEE without it calling any *socket*-related functions.

5 Challenge 4: Bonus

For the last challenge you are free to symbolically execute or fuzz **any** program of your choosing. If you need some guidance or are stuck, you are free to pose questions on the Toledo discussion board or send an email to jeroen.robben@kuleuven.be. This challenge is **optional** but can be used to "recover" some lost points (see Section 6.6 for more info).

6 Report

After performing the above exercises, you have to write a report where you describe your solution(s) and answer the following questions. Every numbered question can be answered in one or two paragraphs and must be answered in your own words. We also give an indication of how many points, out of 23, can be earned for each question (this exact point allocation is tentative).

6.1 Fuzzing (6 points)

1. How does AFL detect interesting new inputs?

2. AFL maintains a set of interesting inputs. How does AFL manage this input set? That is, how does it select the next input to mutate and feed to the program?

3. Why does AFL work better when having access to the source code of the program?

4. Not every bug results in a crash, e.g., a buffer overflow within a stack frame or reading from uninitialised memory. Give a high-level explanation of what needs to be done to make such bugs observable during fuzzing.

5. Which compiler flags and/or protections exist for detecting memory corruption during runtime?

6.2 Symbolic execution (4 points)

1. Give a brief example where AFL would have difficulty exploring a branch but KLEE wouldn't. Explain why.

2. KLEE maintains a set of program states that still need to be further explored. How does KLEE select the next program state to continue exploring?

6.3 Challenge 1 (4 points)

1. Give all terminal commands you used to run AFL. For every bug found, provide an input that was reported by AFL to cause an error.

2. Give all terminal commands you used to run KLEE. For every bug found, provide an input that was reported by KLEE to cause an error.

3. Explain the bug in *logmein* that you've found.

4. Bonus: The developer of *logmein* made the mistake of using a self-designed cryptographic primitive, which is very bad practice in terms of security. Can you find the password that is a pre-image of the hardcoded hash?
   Hint: KLEE also reports an error if it can falsify an *assert()* statement!

6.4 Challenge 2 (5 points)

1. Include the code you had to write in order to fuzz YAJL with AFL in the submitted ZIP file. In the report, describe the commands you had to execute so you could fuzz YAJL with AFL.

2. If you didn't do this in the previous step, include the code you had to write in order to fuzz the JSON parser in AFL's *persistent mode*. How does persistent mode differ from non-persistent mode? Why is it faster?

3. Explain which test cases / seeds you used for fuzzing YAJL with AFL. Include them in the submitted ZIP file.

4. What input was reported by AFL to cause an error? At which line of code does the bug occur? What kind of bug is it (buffer overflow, double-free, integer overflow, or something else)?

6.5 Challenge 3 (4 points)

1. Which coding construct slowed down the symbolic execution process? Do you know *why* this caused a slowdown?

2. Which *program input* caused the bug to occur? Include all code that you wrote or edited in the submitted ZIP file.

3. Can you explain the mistake the developer made?

6.6 Challenge 4 (optional)

Recall that the following challenge is optional, meaning you can get a perfect score for the project without solving it. By solving this question though, you can **recover a maximum of two points**. In other words, if you lost two points when answering other questions, but you gave a decent answer to this optional challenge, you will still get a perfect score for the project.

1. Which program did you test?

2. Give an overview of the steps you took to test the chosen program. Did you need to write a harness?

3. Explain how you chose example inputs. If applicable, describe any special findings or results from fuzzing the program.

7 Practicalities

7.1 Required execution time

On our machine, all mentioned vulnerabilities could be found by KLEE / AFL++ in less than 10 minutes.

7.2 Submission format

Please provide your report as a single PDF along with your code in a single ZIP-file.

7.3 Submission deadline

Please submit your solution the latest on **5 December 2023 before 23:59** through Toledo.

7.4 Questions

In case of questions, pose your question in the Toledo discussion board. In case asking your question might leak an answer you can instead send an email to jeroen.robben@kuleuven.be with your question.

If you have hardware issues, such as having insufficient disk space, Docker not being supported your machine, etc, you can contact us. If the issue cannot be solved, we can provide access to a remote virtual machine as a backup solution. Remember to periodically delete the output directories generated by KLEE.

Good luck!

Mathy Vanhoef
Jeroen Robben