CISPA
HELMHOLTZ CENTER FOR INFORMATION SECURITY

Security Testing
WS 2023/2024

Prof. Dr. Andreas Zeller
Leon Bettscheider
José Antonio Zamudio Amaya

Project 1 (Version 3)

Due: 02. January 2024

Important update (13.12.2023)

Your fuzzer will be stopped as soon as 100.000 inputs or 30 minutes of execution time are reached.

- Example 1: Your fuzzer generated only 20.000 inputs after 30 minutes. We will stop it after 30 minutes, and use the coverage achieved with these 20.000 inputs.
- Example 2: Your fuzzer generated 100.000 inputs after 10 minutes. We will use the coverage achieved with these 100.000 inputs.

> The lecture is based on The Fuzzing Book, an *interactive textbook that allows you to try out code right in your web browser*.
> The Fuzzing Book code is additionally available as a Python pip package. To work on the exercises, please install the package locally:
> `pip3 install fuzzingbook`

Project Description

For this project, you will build a **grammar-based blackbox fuzzer** to test SQLite, which is the most widely deployed SQL database engine in the world.

IMPORTANT: The page at https://www.sqlite.org/lang.html lists all commands understood by SQLite. This page should guide your grammar implementation.

SQLite is implemented in C and for this project, you will work on Linux.

We provide you with a **Dockerfile** that sets up the project framework in a Ubuntu environment, including a GUI delivered to your web browser. This allows all students to work on the same platform, independently of their host setup (Linux, Windows, MacOS).

Setting up the Docker Framework

Docker is one of the most popular containerisation software and has become one of the most widely used industry standards. In this project, Docker will serve to encapsulate all project dependencies as well as provide the same operational layer for all students. If you do not have Docker installed on your computer, refer to the get-started guide Docker provides.

Once you have installed Docker, you can get started with your project following these steps:

1. **Build the Docker Image**:

First, you need to build the Docker image using the provided Dockerfile. Open your terminal and navigate to the directory containing the Dockerfile and the project1 folder. Use the following command to build the image.

```
docker build -t sectest .
```

The `-t` flag allows you to specify a name (in this case, "sectest") for the image.

2. **Run the Docker Container**:

   Once the image is built, you can run the Docker container with the following command:

   ```
   docker run -d --name sectest_container -v sectest:/app -p 6080:80 sectest
   ```

   This command maps port 6080 on your host to port 80 inside the container and names the container "sectest_container", using the "sectest" image.

   The `-d` flag runs the container in detached mode, allowing you to use the terminal for other tasks.

   The `--name` flag sets up a name for the container.

   The `-v` flag creates and maps the volume named "sectest" on the host system to the directory "/app" inside the Docker container.

3. **Stop and restart the Docker Container**

   If you want to stop a running container without losing progress, use the docker stop command, specifying the container's name or ID:

   ```
   docker stop sectest_container
   ```

   If you've stopped the container and want to restart it, use the docker start command. This command will ensure the same volume is executed, so you will not lose the progress you have made:

   ```
   docker start sectest_container
   ```

4. **Accessing the GUI and Tools**:

   After starting the Docker container, you can access the GUI (http://localhost:6080/) and various tools as follows:

   - **Terminal**: Open a terminal within the Docker environment by navigating to "Start," "System Tools," and selecting "LXTerminal."
   - **File Explorer**: Open the file explorer by going to "Start", "System Tools", and selecting "File Manager PCManFM"
   - **Web Browser**: Open your web browser by going to "Start," "Internet," and selecting either Mozilla Firefox or Google Chrome.
   - **JupyterLab**: To work with Jupyter notebooks, you can open a terminal within the Docker environment and run the following command:

     ```
     python3.10 -m jupyterlab --allow-root
     ```

     This command will start JupyterLab, and you can access it through your web browser by visiting the URL provided in the terminal.

Now you are ready to begin your project, utilizing the Docker container for a consistent development environment. If you have any question about the deployment, please ask on Askbot. When it's time for you to submit, you can upload your ZIP to the CMS using the interface provided by this Docker. Simply access your preferred web browser, log into CMS and upload your submission.

Fuzzing Framework (inside the Docker container)

The fuzzing framework is stored at `/root/project1`.

The files in this directory are listed below.

The files marked with **[C]** should be changed by you.

The files marked with **[X]** must not be changed.

```
.
├── [C] fuzzer.py
├── [C] grammar.py
├── [X] Makefile
├── [X] run.py
├── [X] shell.c
├── [X] sqlite3
├── [X] sqlite3.c
├── [X] sqlite3ext.h
├── [X] sqlite3.h
├── [X] sqlite3-shell.gcno
└── [X] sqlite3-sqlite3.gcno
```

The main logic is implemented in **run.py**. This file *must not be changed*.

To generate an input, **run.py** calls the function **fuzz_one_input** in **fuzzer.py**.

fuzzer.py uses the grammar provided in **grammar.py**.

You can run **run.py** as follows to generate 1000 inputs, run them sequentially in SQLite, and do one final branch coverage measurement:

```
python3.10 run.py 1000
```

You can then find the coverage information in `coverage_report.csv`.

Also, there is a coverage plot generated in `plot.pdf`.

With the current arguments, you will only see a constant line.

If you want to observe the coverage progression, use the following arguments:

```
python3.10 run.py --plot-every-x 100 1000
```

This will do a coverage measurement every 100 inputs and generate 1000 inputs in total. You can now observe the coverage progression in `plot.pdf`. Note that you first have to implement an interesting grammar to achieve non-constant coverage.

If you want to get a more detailed coverage report in `coverage_report.html`, which might provide helpful insights to improve your grammar and implementation, use the following command after running `run.py`:

```
make coverage-html
```

You may only change the grammar.py and fuzzer.py files.

Project Tasks

There are two tasks in this project:

- Implement a diverse SQLite grammar in **grammar.py** which is able to generate all commands understood by SQLite. The grammar should be general: For instance, a `CREATE TABLE` command should be able to generate diverse table names. The page at

https://www.sqlite.org/lang.html provides very detailed information about all commands of SQLite.

- Implement the function **fuzz_one_input** in **fuzzer.py**. The signature of this function must not be changed. The function should implement a grammar-based blackbox input generation for SQLite. You may add arbitrary code and functions in this file, but the entry point must be **fuzz_one_input**. Be creative in this task, and come up with ways to generate interesting inputs!

You may use any fuzzers provided by FuzzingBook or implemented by yourself in Python. Other (third-party) fuzzers or programming languages are not allowed.

To get you started, find a grammar below that produces the `CREATE TABLE` command. Note that this grammar does not capture all details of the `CREATE TABLE` command.

In [ ]:
```
grammar = {
"<start>":              ["<create_table>"],
"<create_table>":       ["CREATE TABLE <table_name> (<table_columns_def>);"],
"<table_name>":         ["<string>"],
"<table_columns_def>":["<table_column_def>",
                        "<table_columns_def>,<table_column_def>"],
"<table_column_def>":  ["<string> TEXT"],
"<string>":             ["<letter>", "<letter><string>"],
"<letter>":             ["a", "b", "c", ..., "z"]
}
```

How we evaluate your project

- We grade your project across three dimensions that you find in the **Evaluation Guidelines** at the end of this document.
- The most important dimension is `branch coverage`.
- To compute the branch coverage, we fuzz SQLite with 100.000 inputs generated by your fuzzer: `python3.10 run.py 100000`.
- **There is a hard limit of 30 minutes of execution time for your fuzzer. The evaluation will be stopped as soon as 30 minutes, or 100.000 inputs are reached. The first metric to be reached stops the evaluation.**
  - Example 1: Your fuzzer generated only 20.000 inputs after 30 minutes. We will stop it after 30 minutes, and use the coverage achieved with these 20.000 inputs.
  - Example 2: Your fuzzer generated 100.000 inputs after 10 minutes. We will use the coverage achieved with these 100.000 inputs.
- Then we measure the `branch_percent` coverage in `sqlite3.c` as shown in the table below.

Below you find an example `coverage-report.csv` with the relevant branch coverage of `sqlite3.c` highlighted.

filename	line_total	line_covered	line_percent	branch_total	branch_covered	branch_per
shell.c	8645	575	0.067	6458	259	0.04
sqlite3.c	45901	12443	0.271	30925	5818	***0.188***

Tips to improve your project

After you run `python3.10 run.py 100000`, two files are generated:

- `plot.pdf`, shows the temporal branch coverage progression. This is only for your information and might help you to guide your implementation.

- `coverage-report.csv`, will be used by us to grade your project. We will focus only on the **branch coverage** in **sqlite3.c**.

If you run `make coverage-html` after running `run.py`, another file is generated:

- `coverage-report.html`, which provides a much more detailed coverage report, that allows you to investigate the code your fuzzer covered in detail and might help you to improve your implementation.

As a suggestion, you could generate just one input:

```
python3.10 run.py 1
```

Then produce a html coverage report:

```
make coverage-html
```

And check out which lines are not covered by this one input. Maybe you can tweak the grammar a little bit such that neighboring lines are covered? To find out which input your fuzzer generated, you might want to print the generated input in `fuzz_one_input`.

Notes

- Converting an extensive SQLite trace to a context-free grammar and using this as your grammar is not a valid solution. Your fuzzer should be able to come up with a large set of different combinations of SQL commands.
- Each fuzzing run starts with an empty database. However, the database persists during the execution of the 100.000 inputs.
- Fuzzing SQLite is difficult because *syntactically valid inputs may be rejected when they are incompatible with the current database state*. For instance, you can only delete a table if it was created before. Use this information in your implementation of `fuzz_one_input`.
- It is not sufficient to only implement a grammar in `grammar.py`. You must also implement a fuzzer in `fuzzer.py`.
- **fuzz_one_input** should generate only one SQLite command. It's not allowed to join multiple commands with a semicolon and return them as one command.
- The start symbol of your grammar must be **<start>**.

Evaluation Guidelines

You should work individually on this project. Group work is not permitted.

Your fuzzer will be evaluated across three dimensions and needs at least a combined **50%** over all dimensions to pass the project. The total sum of all the scores obtained in the 3 dimensions should be at least 5 or higher. (E.G. 2/5, 1/3, 2/2 means PASS)

1. Branch coverage in sqlite3.c (5 points)

We will measure how much branch coverage your fuzzer achieves in **sqlite3.c**. During the evaluation, we will use your fuzzer to generate 100.000 inputs and measure the branch coverage achieved in SQLite. We will repeat this measurement five times and use the mean branch coverage.

This part contributes to your final project grade as follows: `5 * (your_mean_branch_coverage / goal_branch_coverage)`

2. Bug finding capability (3 points)

If your fuzzer is able to trigger an error or crash in SQLite, you will get extra points. Note that finding a bug in SQLite might be difficult. Hence, when we grade your submission, we will introduce several bugs into the SQLite implementation and evaluate the ability of your fuzzer to discover them.

During this measurement, we will seed one bug at a time, and check whether your fuzzer can trigger it with a budget of generated inputs.

This part contributes to your final project grade as follows: `3 * (#your_found_bugs / #goal_found_bugs)`

3. Input diversity (2 points)

We will assess how diverse the inputs that your fuzzer generates are. To measure this, we will define a number of interesting `input features` (e.g. a specific edge case of the `CREATE TABLE` command), and check if your fuzzer can generate an input that includes this feature, with a budget of 100.000 generated inputs.

This part contributes to your final project grade as follows: `2 * (#your_interesting_input_features / #goal_interesting_input_features)`

Guaranteed passing criterium

If you do not pass by our evaluation guidelines, you are still guaranteed to pass the project if your fuzzer achieves at least **30%** branch coverage (as measured by our framework) in sqlite3.c.

Submission format

Submit your solution as a ZIP file on your status page in the CMS.

Your solution should only contain the `fuzzer.py` and `grammar.py` files you implemented.

No other file may be changed.

Double-check that your submission is runnable. We cannot grade submissions that are not runnable or have bugs.