Artifact Abstract: ASanity: On Bug Shadowing by Early ASan Exits

Vincent Ulitzsch
Technische Universität Berlin - SECT
Berlin, Germany
vincent@sect.tu-berlin.de

Deniz Scholz
Technische Universität Berlin - SECT
Berlin, Germany
notdenizscholz@gmail.com

Dominik Maier
Technische Universität Berlin - SECT
Berlin, Germany
dmaier@sect.tu-berlin.de

I. Artifact Evaluation – Goals

The goal of the artifact evaluation for *ASanity: On Bug Shadowing by Early ASan Exits* is two-fold:

- To confirm that the data analysis was correct
- To confirm that the data collected was correct, namely the scraping of OSS-Fuzz issues and re-running them without the `-fsanitize=recover` flag.

The artifact folder contains the dataset used for evaluation (`artifact_evaluation_new.zip`) as well as a VM .ova file, `reproduction_ubuntu.ova`), that can be used to run the scripts. The artifact inside the VM is divided into two parts.

- `data_collection` All scripts necessary for the data collection
- `data_analysis` The scripts used to analyze the collected data

To start the evaluation, import and the launch the Ubuntu-VM and follow the steps below.

1) Launch the VM, log in to user `test` with password `test`.
2) Open a terminal and change to root user via `su`, the password is again `test`.
3) Change into the artifact evaluation directory

```
cd /home/test/artifact_evaluation/
```

We will now walk you through each evaluation step.

II. Artifact Evaluation – Analyzing the Data

Our dataset compromises the following elements:

- data_collection/monorail-scraper/scraped_issues — Multiple json files, containing a list of objects, each describing one issue scraped from oss-fuzz.
- data_collection/filtered_issues — Multiple json files, containing a list of objects, each describing one issue scraped from oss-fuzz. We only retain issues that were reported as heap OOB-R, however.
- data_analysis/dataset — A collection of ASAN error logs for each issue that we attempted to reproduce, i.e., each issue from `scraped_issues`. The structure of the directory as follows. We store the logs for an issue in `dataset/project-name/issue/`. The file `old.txt` is the ASAN error log produced by running the target on the crashing testcase, *without* disabling ASAN early exits and using a prebuilt binary, downloaded from cloud. The file `new.txt` is the ASAN error log with disabled ASAN early exits, running on a re-compiled binary.

To assemble the statistics that we present in the paper, simply run our script `analyse_errors.py`. The script will iterate through all ASAN error logs and parse them. We track the issue in our statistics if the ASAN error log of the run with enabled early-exists reports an OOB-R, but the error log of the run with disabled early-exists additionally reports an OOB-W or use-after-free, . The script outputs a table in latex format and summarizes a few statistics.

```
[root@5f91d5a19ccf /]# cd data_analysis
[root@5f91d5a19ccf /]# python3 analyse_errors.py
[...]
\begin{tabular}{lrrr}
\toprule
   Projects &  oobr &  oobw &  uaf \\
\midrule
   libdwarf &     1 &     1 &    0 \\
[...]
\bottomrule
\end{tabular}
38 out of 750 out-of-bounds read also triggered an
    out-of-bounds write or use-after-free
9 out of 750 out-of-bounds read also triggered an
    use-after-free
30 out of 750 out-of-bounds read also triggered an
    out-of-bounds write
```

This should match the numbers reported in the paper.

III. Artifact Evaluation – Collecting the Dataset

The aim of this artifact evaluation step is confirm that the data was collected in a correct way. Recall that for the ASanity paper, we perform the following steps:

1) Scraped the issue list from OSS-Fuzz
2) Filtered issues for out of bounds read issues
3) Reran the pre-compiled targets against the respective testcases
4) Recompiled the targets at the vulnerable commit with `-fsanitize=recover` option disabled
5) Parsed the ASAN error logs to see which OOB-R turned out to be OOB-Writes or use-after-free issues as well.

A. Issue Scraping OSS-Fuzz

The OSS-Fuzz Project offers an extensive collection of bugs from open-source projects. This collection is publicly visible and can be found at OSS-Fuzz Monorail [1]. The OSS-Fuzz Monorail scraper includes a large table of reported issues. The table columns include a unique internal ID, the type of the bug, a component, the status, the project, the date of the report, the owner, and a summary with labels. We build on top of the Monorail Scraper. The scraper can be started by running the python3 scrape_oss_fuzz_issue_range .py [–h] –s START –e END command in a command line interface. By setting the START and END values, we can specify the range of ids of the projects which will be scraped. For example, the python3 scrape_oss_fuzz_issue_range .py –s 10000 –e 20000 will scrape all issues with ID's between 10000 and 20000. The results will be structured in the JSON format, written to the standard output, and printed in the command line interface. We can pipe the output from stdout to a file by adding > filename.json to the end of the command, which results in the following command:

```
[root@5f91d5a19ccf]# cd data_collection/monorail-scraper
[root@5f91d5a19ccf monorail-scraper]# python3 scrape_oss_fuzz_issue_range.py -s START -e END > START_END.json
```

For our paper, we executed the command multiple times with smaller ranges, resulting in multiple JSON files. Each JSON file contains a list of objects representing an issue. Find the results of our run in `data_collection/monorail-scraper/scraped_issues`.

B. Filtering Issues For OOB-R

Next, we want to filter to scraped issues to only retain issues that are OOB-R. To this end, we can call

```
[root@5f91d5a19ccf monorail-scraper]# python3 filter.py
```

This will store the remaining issues in the directory `filtered_issues`.

C. Reproducing the Issue and Rerunning

As a last step, we run the script main.py

```
[root@5f91d5a19ccf data_collection]# python3 main.py
```

As a pre-requisite, you might have to run `gcloud init`, so that `gcloud` cli can fetch binaries. This, iterates through each issue in filtered_issues and:

1) Downloads the pre-compiled binary of the project from gcloud.
2) Reruns the binary with the crashing testcase to collect an ASan log. ASanity prints and records the ASAN output in `data_analysis/dataset/<project>/<issue-id>/old.txt`.
3) Pulls the project at the commit that is specified in the issue description.
4) Recompiled the project with `-fsanitize=recover` flag enabled.
5) Reruns the binary with the crashing testcase to collect an ASan log of the relaxed ASan run. The output of this ASAN run is printed and recorded in `data_analysis/dataset/<project>/<issue-id>/new.txt`.

IV. Building your own VM

To launch your own VM, the following steps are needed:

1) Install the package dependencies via apt

```
$ apt-get update -y && apt-get install -y python3 pip gcc python3-dev sudo curl wget
$ apt-get -y install apt-transport-https \
    ca-certificates \
    curl \
    gnupg2 \
    software-properties-common
```

2) Install the Python dependencies as listed in the requirements file in `artifact_evaluation/requirements.txt`

```
$ pip3 install -r requirements.txt
```

3) Install google chrome version 95.06, as provided with the artifact evaluation directory:

```
$ dpkg -i data_collection/monorail-scraper/google-chrome-stable_95.0.4638.69-1_amd64.deb; apt-get -y -f install
```

4) Install google-cloud-sdk and docker, as described in https://cloud.google.com/sdk/docs/install and https://docs.docker.com/engine/install/
5) Init google cloud with an auth token

```
gcloud init
```

6) Add the monorail-scaper to your path

```
$ export PATH=$PATH:(realpath data_collection/monorail-scraper)
```

Your VM is now prepared to reproduce the artifact evaluation as described above.