A Artifact Appendix

A.1 Abstract

Our artifact includes the regexps processed in this paper, a record of the NPM packages analyzed, the REGULATOR workflow and source code, the results produced by both our tool and those used for comparison, and software for computing values and figures found in this paper.

We provide a x86-64 docker container with all prerequisites necessary to compile REGULATOR. A pre-compiled version is also included.

Our results can be validated by re-running the tool to detect and verify ReDoS-vulnerable regexps.

A.2 Artifact check-list (meta-information)

- **Data set:** We use two different datasets in our paper. The first (called Base Dataset in our paper) was sourced from three different collections used in previous ReDoS research (known as Corpus, RegexLib and Snort). The second (called NPM dataset) was instead created during our research, by scraping and extracting the regular expression used in the 10,000 most popular NPM packages. Both datasets are included in the docker container under `/artifacts/data/regex.csv`.
- **Run-time environment:** The software is evaluated using Python 3.8, NodeJS 10.19.0, Postgresql 12, and gcc 9.3.0, running on Ubuntu 20.04 in Docker 20.10.7.
- **Run-time state:** Regulator is based on fuzzing, so results might slightly differ between each run.
- **Execution:** Since Regulator fuzzes each regular expression for a given amount of time, we recommend to run the tool on a machine with no significant background tasks.
- **Metrics:** Each tool evaluated in our paper reports whether a regular expression is vulnerable to ReDoS. Our artifact reports the following metrics: true positives, false positive and false negatives of each tools (Table 3 and Table 4 of our paper).
- **Output:** For each regular expression, a record is produced of the fuzz witness, the classified growth-function (if super-linear), whether it was verified to cause significant slow-down, and the minimum string-length length required to achieve that slow-down.
- **Experiments:** Included are setups for these experiments: running REGULATOR against regular expressions, and running the comparison tools REGULATOR-PerfFuzz, ReScue, NFAA, Revealer, RXXR2, and Rexploiter against regular expressions.
- **How much disk space required (approximately)?:** Approximately 10 GB of disk space is required.
- **How much time is needed to prepare workflow (approximately)?:** About 30 minutes is required to laod the docker container, start services, and queue regexps in the workflow.
- **How much time is needed to complete experiments (approximately)?:** The experiments require approximately 10,000 CPU-hours. The workflow can be configured to make use of multiple CPU cores at once, to reduce the wall-clock time required.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** MIT
- **Archived (provide DOI)?:** 10.5281/zenodo.5669243

A.3 Description

A.3.1 How to access

The docker container where all artifacts are stored can be downloaded at the following url: https://doi.org/10.5281/zenodo.5669243

A.3.2 Hardware dependencies

Regulator does not have any particular hardware dependency (the docker container was tested on a x86-64 Linux system), but we recommend to using a server with a substantial number of CPU cores.

A.3.3 Software dependencies

The only software dependency is `docker` (tested on version 20.10.7), plus a `gzip` decompression utility.

A.4 Installation

The artifact contains a docker container. After downloading the compressed image `regulator_artifacts.tar.gz`, decompress it using a decompression tool. For example, on most Linux systems: `gzip -d regulator_artifacts.tar.gz` to produce the file `regulator_artifacts.tar`. It can be loaded into docker with the following command `docker load < regulator_artifacts.tar`. Then, the container can be started with `docker run -it regulator_artifacts /bin/bash`.

A.5 Experiment workflow

The core results from our paper can be reproduced by running REGULATOR and previous research tools against the Base and NPM dataset. The previous tools tested in this paper are: RXXR2, Rexploiter, NFAA, ReScue, PerfFuzz, Revealer. The first four tools were packaged by Davis Jamie in the `vuln-regex-detector` project [1]. In the following sections we therefore present 4 different workflows to run REGULATOR, PerfFuzz, Revealer, and vuln-regex-detector.

REGULATOR. The experiment workflow to run REGULATOR is documented in `/artifacts/detectors/regulator/README.md`. Before running our tool, the regular expressions must be loaded inside a postgres database using the `add_to_queue.py` script.

REGULATOR has then three phases:

[1] https://github.com/davisjam/vuln-regex-detector

1. Fuzzing Stage: the target regular expression is fuzzed. The output of this step is a witness string, i.e. the string that has the highest number of executed bytecode instructions. This step is implemented in the `fuzz_from_queue.py` script.

2. Pumping Stage: the witness string is translated in a pump formula. This step is implemented in the `pump_all.py` script.

3. Dynamic Validation: the pump formula is tested against the irregexp engine. If the formula causes a slowdown of more than 10 seconds then the target regular expression is marked as vulnerable to ReDoS. This step is implemented in the `binsearch_pump.py` script.

PerfFuzz. The workflow to run PerfFuzz is quite similar to REGULATOR's, and more instructions can be found under `/artifacts/detectors/regulator/README.md`.

Revealer. The workflow to run Revealer is documented in `/artifacts/detectors/revealer/README.regulator.md`. To run this tool, invoke the script `run_with_timeout.py`.

vuln-regex-detector. The workflow to run ReScue, RXXR2, NFAA, and Rexploiter. This is documented in `/artifacts/detectors/davis-detectors/README.md`.

A.6 Evaluation and expected results

In this paper we show that REGULATOR outperforms previous ReDoS detectors. This is the core result of this research, and is shown in Table 3 and Table 4 of the paper. Rerunning the tool should show more true positive detections than prior work.

There are two ways to reproduce these results. The first is to re-use the output of our experiments (stored under `artifacts/data/`, the second is to run the workflows discussed in the previous section and copy these newly generated results into `/artifacts/data`. See the `README` for each workflow for more details.

In both cases, the script `/artifacts/scripts/analyze_results.py` will summarize the results and produce the numbers contained in Table 3 and Table 4.

If running the entire workflow requires too many resources, REGULATOR can be more quickly evaluated by running the workflow for a random subset of regexps which were reported as vulnerable in this work. We expect a high percentage (at least 80%) to be reproducible.