USENIX'23 Artifact Appendix: Silent Spring: Prototype Pollution Leads to Remote Code Execution in Node.js

Mikhail Shcherbakov
KTH Royal Institute of Technology

Musard Balliu
KTH Royal Institute of Technology

Cristian-Alexandru Staicu
CISPA Helmholtz Center for Information Security

A Artifact Appendix

A.1 Abstract

This artifact implements static code analysis for detecting prototype pollution vulnerabilities and gadgets in server-side JavaScript libraries and applications, including the Node.js source code. The analysis builds on GitHub's CodeQL framework to identify prototype pollution sinks and gadgets. We evaluate precision and recall metrics for prototype pollution detection in comparison with existing CodeQL analysis as well as the tool ODGen. Further, we evaluate the capabilities of our tool, in combination with dynamic analysis, to detect gadgets in a range of popular applications, including the Node.js source code. Finally, we evaluate the prevalence of detected gadgets on a dataset of popular libraries. All of the artifact evaluation results refer to Section 6 of the paper and the Appendix. The artifact evaluation aims for the three badges: available, functional, and reproducible.

A.2 Description & Requirements

Here we describe hardware and software requirements to run the artifact, as well as an overview of the benchmarks.

A.2.1 Security, privacy, and ethical concerns

There are no risks for the reviewers relating to security and privacy of their machines. The artifact has been used to detect 8 remote code execution vulnerabilities in production-ready applications and these vulnerabilities have been responsibly disclosed to the vendors. We do not provide any details on exploits that are yet to be fixed by the developers. Moreover, exploit generation is a manual process, hence it is not part of this artifact evaluation.

A.2.2 How to access

The artifact is accessible on GitHub at address https://github.com/yuske/silent-spring/tree/2c7cfab. The reproducibility of the results is supported by two modes: (1) a prepackaged docker container and (2) detailed instructions on how to set up the environment on own machine.

A.2.3 Hardware dependencies

We perform the experiments on an Intel Core i7-8850H CPU 2.60GHz, 16 GB RAM, and 50 GB of disk space. No specific hardware features are required for the artifact evaluation.

A.2.4 Software dependencies

We originally run our experiments (except for the experiment **E2** of ODGen evaluation) on Windows OS and presented these results in the paper. However, CodeQL and our evaluation scripts support Linux and provide similar results.

A.2.5 Benchmarks

We provide five benchmarks for our experiments. The root directory of the artifact repository contains folders with benchmark names from the list below. Clone the repository with its Git submodules and follow the instructions of Appendix A.3 to download all code of benchmark-silent-spring and benchmark-npm-packages.

(benchmark-silent-spring): We compile an open-source dataset of 100 vulnerable Node.js packages to evaluate the recall and precision metrics of our static analysis. We refer to Section 6.1 and Table 3 of the paper for details of the benchmark and our experiments against this set of packages.

(benchmark-odgen): We consider the dataset of 19 packages provided by the tool ODGen to compare our static analysis approach with the state-of-the-art results of ODGen. The paper presents the details of the dataset and analysis results in Section 6.1 and Table 3 as well.

(benchmark-popular-apps): We evaluate our approach on popular Node.js applications from GitHub. The benchmark contains exact versions of 15 analyzed applications.

The evaluation results are presented in Section 6.3 and Table 2.

(benchmark-nodejs): We run our gadget detection analysis against Node.js version 16.13.1. The source code of the analyzed Node.js is located in a folder of the benchmark. Table 1 of the paper reports all the detected gadgets and their summary.

(benchmark-npm-packages): We estimate the prevalence of the gadgets in an experiment with the 10,000 most dependent-upon NPM packages. This benchmark contains these NPM packages. We describe the results of the experiment in the last paragraph of Section 6.2.3.

A.3 Set-up

We provide two modes for testing the artifacts (1) a docker image with the prepared environment and (2) detailed instructions on how to set up the environment on own machine. To use the docker image, pull the docker image `yu5k3/silent-spring-experiments:latest` from Docker Hub, launch a docker container, and run `/bin/bash` into the container to get access to the pre-configured environment. In this mode, the reviewers may skip the setup and installation steps, and move directly to the folder `~/projs/silent-spring` in the docker container and follow the instructions from Appendix A.3.2.

The following steps describe how to set up a required environment on own machine.

(S1): Clone the ODGen repository https://github.com/Song-Li/ODGen.git and checkout commit `306f6f2`. Follow its README file to set up the tool.

(S2): Clone the Silent Spring repository with its submodules https://github.com/yuske/silent-spring.git and checkout commit `2c7cfab`.

(S3): Move to the scripts by `cd silent-spring/scripts/`. Further, it is important to run any setup and evaluation scripts using the `scripts` as a working directory.

(S4): Run the script `./benchmark-silent-spring.install-dependencies.sh` to install dependencies for benchmark-silent-spring.

(S5): Install NPM dependencies for the scripts by `npm i`.

A.3.1 Installation

The experimental evaluation requires the following software:

(I1): Node.js v.16.13.1. Follow the instruction on the official website to install Node.js.

(I2): Cloc. We use `cloc` application to count lines of analyzed code. Use in the official repository to download and install the latest version.

(I3): CodeQL v.2.9.2. Download and unzip an asset for your platform of the version 2.9.2 from the official repository. Add the path of the `codeql` folder to `PATH` environment variable.

A.3.2 Basic Test

We recommend a basic test for 1-2 NPM packages with our CodeQL queries to check that all required components function correctly. The execution of command `node ./benchmark-silent-spring.codeql.js -l 1` from directory `scripts` performs the analysis of only one NPM package from benchmark-silent-spring and stores the results at `../raw-data/benchmark-silent-spring.codeql.limit.md`. The analysis should be completed in about 3 minutes. We provide a reference file for comparison with the basic test results. The easiest way to compare the evaluation results with the reference is to execute `git diff -- ../raw-data/benchmark-silent-spring.codeql.limit.md`. The count of detected cases in the table should be the same.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Our static analysis tool, built on top of CodeQL, achieves higher recall (up to 97%) for prototype pollution detection as compared to existing CodeQL analysis and the state-of-the-art tool ODGen. At the same time, it achieves moderate precision (on average 39%). This is evaluated by the experiments **(E1)** and **(E2)** described in Section 6.1 of the paper with results reported in Table 3.

(C2): Our tool has been used to uncover 8 new critical vulnerabilities in popular Node.js open-source applications. This is evaluated by the experiment **(E3)** and described in Section 6.3 and Table 2 of the paper.

(C3): We use static and dynamic analysis to detect 11 new gadgets in Node.js code that may lead to Remote Code Execution attacks. The gadget detection is evaluated by the experiments **(E4)** and **(E5)** described in Section 6.2 and summarized in Table 1 of the paper.

(C4): We estimate the prevalence of the detected gadgets on 10,000 most dependent-upon NPM packages. The measurement of the prevalence is shown by the experiment **(E6)** and described in Section 6.2.3 of the paper.

A.4.2 Experiments

All experiments should be run in the `scripts` folder to match the relative paths in the script files. All scripts collect the results of experiments in the folder `raw-data`. This folder already contains our results which can be used as reference for comparison.

(E1): Prototype pollution detection with CodeQL [1 human-hour + 3 compute-hours]: evaluate the existing CodeQL analysis and our analysis framework on benchmark-silent-spring and benchmark-odgen.
Execution: Run the following scripts:

```
>node ./benchmark-silent-spring.codeql.js
>node ./benchmark-silent-spring.baseline.
    codeql.js
>node ./benchmark-odgen.codeql.js
```

Results: The file names of the analysis results correspond to the file names with `.md` extension. The files consist of tables where *columns* contain the detected cases for the executed CodeQL queries. The last row calculates the total number of True Positives (TP) and False Positives (FP), as well as the recall and precision metrics. The result for benchmark-odgen contains only detected *sinks* that should be matched to code locations from `.PoC*.expected` files (including `.PoC.ext.expected`), e.g., `benchmark-odgen/asciitable.js@1.0.2/asciitable.PoC.expected`. We summarized benchmark-silent-spring results in Table 3 in the paper. The experiment should yield the recall and precision metrics that correspond to the metrics of *Total* row in Table 3. The results of benchmark-odgen are discussed in the last paragraph of Section 6.1.

(E2): Prototype pollution detection by ODGen [1 human-hour + 11 compute-hours]: evaluate ODGen analysis on benchmark-silent-spring and benchmark-odgen.
Preparation: Set the absolute paths to ODGen (variable `odgenDir`) and the silent-spring folder (variable `ppStuffDir`) in `benchmark-odgen.odgen.js` and `benchmark-silent-spring.odgen.js` files. This is already done for the provided docker image.
Execution: Run the following scripts:

```
>node ./benchmark-silent-spring.odgen.js
>node ./benchmark-odgen.odgen.js
```

Results: The scripts create two reports for benchmark-silent-spring and benchmark-odgen that are structured as the results of **(E1)**. The results in `benchmark-silent-spring.odgen.md` have worse metrics than we reported. This is because ODGen makes random choices and, in our experiments, we ran the ODGen tool several times and merged their best results from all runs in Table 2 (in order to compare with their best configuration).

(E3): Vulnerability detection in applications [1 human-hour]: evaluate our analysis to detect prototype pollution in Node.js applications.
Execution: Run the following script: `node ./benchmark-popular-apps.codeql.js`
Results: File `benchmark-popular-apps.codeql.md` contains the count of the detected prototype pollution cases and links to the source code of the detected sinks. The number of the detected cases corresponds to the column *Total - Cases* of Table 2 in the paper. The provided script reports two extra cases for one *parse-server* and one *sails* due to the usage of earlier version of CodeQL in the original experiments.

(E4): Gadget detection (dynamic analysis phase) [1 human-hour]: evaluate the dynamic analysis of three Node.js APIs for prototype pollution gadgets.
Execution: Run the following scripts:

```
>node ./gadgets.infer-properties.js
>node ./gadgets.dynamic-analysis.js
```

Results: The scripts report undefined properties subject to prototype pollution in the file `gadgets.dynamic-analysis.csv`. We detected 37 undefined property reads in `child_process`, `require`, and `vm` APIs, and described this experiment in Section 6.2.1. The property `TERM` can be reached on Windows but not Linux. The list of the reported properties contains *universal properties* of the identified gadgets that we describe in Table 1 in the paper.

(E5): Gadget detection (static analysis phase) [1 human-hour]: evaluate the data flow analysis for the detected properties in **(E4)**.
Execution: Run the following script: `node ./gadgets.static-analysis.js`
Results: We implement a CodeQL-based analysis to detect flows from polluted properties to sinks, and validate the results manually, as described in Section 6.2.2. The provided script summarizes the results and reports *sources* that are the exported functions triggering a reading of polluted properties and *sinks* that are the internal functions taking the read values. The report `gadgets.static-analysis.md` counts *sources* and *sinks* to show feasibility of the manual analysis. The folder `gadgets.static-analysis.tmp` contains the detected function names.

(E6): Gadgets prevalence estimation [1 human-hour]: analyze the most dependent-upon NPM packages to estimate potential exploitability of detected gadgets.
Preparation: Script `./gadgets.download-packages.sh` downloads NPM packages for analysis (execution takes 40 mins). Skip this step if you use the docker image.
Execution: Run the script (takes about 15 minutes): `node ./gadgets.prevalence-analysis.js`
Results: The last line of the script's output contains analysis results, reporting *Packages with no main - 2041; packages have relative 'require' - 4393; packages have 'child_process' methods - 350.* We report the results of our experiment in the last paragraph of Section 6.2.3 in the paper. The slight discrepancy is due to the use of different versions of the NPM packages for the analysis.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.