USENIX Security '24 Artifact Appendix: GHUNTER: Universal Prototype Pollution Gadgets in JavaScript Runtimes

Eric Cornelissen
KTH Royal Institute of Technology

Mikhail Shcherbakov
KTH Royal Institute of Technology

Musard Balliu
KTH Royal Institute of Technology

A Artifact Appendix

A.1 Abstract

The artifacts develop lightweight taint analysis on top of the JavaScript runtimes Node.js and Deno with the goal of identifying prototype pollution gadgets. In particular, each artifact modifies the V8 JavaScript engine shared by Node.js and Deno as well as some minor aspects of each runtime itself; these changes are present as `.patch` files in the artifact. Additionally, each builds on top of the project with tooling to run our analysis and generate results. Finally, the last artifact constitutes modifications to Silent Spring used for the comparison between GHUNTER and Silent Spring.

We demonstrate the functionality and reproducibility of the analysis artifacts and evaluate the effectiveness of our analysis against Silent Spring in terms of precision and recall. The results of the former refer to Section 5.1 and Table 1, 5 and 6 while the latter refers to Section 5.2 and Table 2 and 3.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

There are no risks for the users relating to security and privacy of their machines. The artifact has been used to detect gadgets in production-ready software and these vulnerabilities have been responsibly disclosed to the vendors.

A.2.2 How to access

The artifacts are accessible on GitHub at github.com/KTH-LangSec/ghunter/tree/23abc11 which encompasses three sub projects: the Deno analysis artifact, the Node.js analysis artifact, and the Silent Spring comparison artifact.

A.2.3 Hardware dependencies

We performed the experiments described in this appendix on an AMD Ryzen 7 3700x 8-core CPU (3.60GHz) with 32 GB RAM and 50 GB of disk space. No specific hardware features are required for the artifact evaluation.

A.2.4 Software dependencies

We performed the experiments on the Ubuntu 22.04 OS. We used Docker as an OCI container runtime.

A.2.5 Benchmarks

Deno v1.37.2 We run our gadget detection analysis against Deno version 1.37.2. The source code of this benchmark is incorporated as git submodules in the `ghunter4deno` sub project (named `deno`, `deno_core`, and `rusty_v8`). Section 5.1 and Table 1 of the paper reports the aggregate number of gadgets detected and Table 6 of the paper reports all the detected first-order gadgets in detail.

Deno standard library v0.204.0 In addition to Deno v1.37.2, we run our gadget detection analysis against the Deno standard library version 0.204.0. The source code of this benchmark is incorporated as a git submodule in the `ghunter4deno` sub project (named `deno_std`). Section 5.1 and Table 1 and 6 also report on this benchmark.

Node.js v21.0.0 We run our gadget detection analysis against Node.js version 21.0.0. The source code of this benchmark is incorporated as a git submodule in the `ghunter4node` sub project (named `node`). Section 5.1 and Table 1 of the paper reports the aggregate number of gadgets detected and Table 5 of the paper reports all the detected first-order gadgets in detail.
Additionally, we run our gadget detection analysis against Node.js version 21.0.0 for a comparison to Silent Spring. The test cases for this comparison are located `src/ss21`. Section 5.2 and Table 3 of the paper reports on the results of this analysis.

Node.js v16.13.1 We run our gadget detection analysis against Node.js version 16.13.1 for a comparison to Silent Spring. The test cases for this comparison are located `src/ss16`. Section 5.2 and Table 2 of the paper reports on the results of this analysis.

Silent Spring We compare our results against those of Silent Spring. We do this on both Node.js v16.13.1 and v21.0.0. Our adaptation of Silent Spring is located in

the `silentspring4ghunter` sub project. This benchmark re-embeds the respective Node.js benchmarks on separate commits (`a6ae944` and `14966b5` resp.). Section 5.2 and Table 2 and 3 (resp.) of the paper report on the results of this analysis.

A.3 Set-up

We provide two modes for testing the Deno and Node.js artifacts (1) a prepared OCI container and (2) instructions on how to set up the environment from scratch. We only provide instructions on how to set up the environment from scratch for the Silent Spring artifact.

(S1): Deno. For the analysis of Deno use either the OCI container image `ghcr.io/kth-langsec/ghunter4deno`[1] by pulling it, launching it, and attaching a shell. Alternatively, build the container image by following the instructions from the README of github.com/KTH-LangSec/ghunter4deno at commit `63a9faa`. In this mode, the users may skip the rest of (**S1**) and (**I1**).
For a local set-up, clone github.com/KTH-LangSec/ghunter4deno with submodules recursively and checkout commit `63a9faa`. Then continue with (**I1**).

(S2): Node.js. For the analysis of Node.js use either the OCI container image by pulling `ghcr.io/kth-langsec/ghunter4node`[2], launching it, and attaching a shell. Alternatively, build the container image by following the instructions from the README of github.com/KTH-LangSec/ghunter4node at commit `86aad7c`. In this mode, the users may skip the rest of (**S2**) and (**I2**).
For a local set-up, clone github.com/KTH-LangSec/ghunter4node with submodules recursively and checkout commit `86aad7c`. Then continue with (**I2**).

(S3): Silent Spring. For the comparison to Silent Spring, clone github.com/KTH-LangSec/silentspring4ghunter with submodules recursively. For the comparison on Node.js v16.13.1 checkout commit a6ae944 and for the comparison on Node.js v21.0.0 checkout commit 14966b5. In either case, continue with (**I3**)-(**I5**).

A.3.1 Installation

(I1): Deno development prerequisites. See github.com/denoland/deno-docs commit `7b4aa84` file `building_from_source.md`.

(I2): Node.js development prerequisites. See github.com/nodejs/node commit `38d0e69` file `BUILDING.md`.

(I3): CodeQL v2.9.2. Download and unzip an asset for your platform of version 2.9.2 from the official repository. Add the path of the codeql folder to PATH environment variable.

(I4): Node.js v16.13.1. Follow the instructions on the official website to install Node.js version 16.13.1 for the comparison on this Node.js version.

(I5): Node.js v21.0.0. Follow the instructions on the official website to install Node.js version 21.0.0 for the comparison on this Node.js version.

A.3.2 Basic Test

(B1): Deno. We recommend running the source-to-sink analysis with a single test case as a basic test. First build using `./make.sh s2s sync`, then run the basic test using `./analyze.sh 2 20 basic-test`. The first command compiles Deno and can take up to an hour, the latter runs a simple analysis that should take about one minute. This is expected to yield about 6 gadget candidates.

(B2): Node.js. We recommend running the source-to-sink analysis with a single test case as a basic test. We provide a script to perform this test, `./nodejs-test-one.sh`. This will build Node.js for the analysis and run the analysis with a single test. This command compiles Node.js, which can take up to an hour, and runs a simple analysis that should take about one minute. This is expected to yield about 10 unique source-to-sink pairs after filtering.

(B3): Silent Spring. Follow the basic test instructions for the original Silent Spring artifact.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Our dynamic analysis tool applied to Deno uncovered 67 universal gadgets. This is evaluated by experiment (**E1**) and described in Section 5.1 and Table 6 of the paper.

(C2): Our dynamic analysis tool applied to Node.js uncovered 56 universal gadgets. This is evaluated by experiment (**E2**) and described in Section 5.1 and Table 5 of the paper.

(C3): Our dynamic analysis tool has higher precision and recall than Silent Spring for finding universal gadgets on two different Node.js versions. This is evaluated by experiment (**E3**)-(**E6**) and described in Section 5.2 of the paper.

A.4.2 Experiments

We describe a total of 6 experiments, 2 related to claims (**C1**) and (**C2**) and 4 related to (**C3**). The former cover the first 3 benchmarks and the latter cover the last 3 benchmarks.

[1] a4c29470545af82a0d8b446e1594ba4e78ad45babdf3af51c72f54fad1c35860
[2] a2b09930d54d652f192d086a91186d5d9d94c14f2deae451b88e563fcb38231a

(E1): Analysis of Deno, 10 human-minutes + <4 compute-hours + 50GB disk: Full analysis of the Deno runtime for universal gadgets.
Set-up: Follow (**S1**).
Preparation: Follow (**B1**).
Execution: Start `./run.sh`, optionally with a number of workers (default 5) and test timeout (default 20s) as `./run.sh <W> <T>`.
Results: The output at the end of the analysis provides a table of which expected gadget candidates from Table 6 of the paper were found as well as the analysis numbers from Section 5.1 (paragraph *Analysis of Deno*); This output can be recomputed by running the `numbers.sh` script *after* the analysis has finished. The exact numbers may differ but are expected to be similar. The folder `_aggregate` will contain the final two SARIF files for manual analysis, which can be compared to the SARIF files in the `results` directory.

(E2): Analysis of Node.js, 10 human-minutes + <4 compute-hours + 50GB disk: Partial analysis of the Node.js runtime for universal gadgets in the `child_process` API.[3]
Set-up: Follow (**S2**).
Preparation: Follow (**B2**).
Execution: Start `./run-child_process-s2s.sh`, optionally with a number of workers (default 5) and test timeout (default 20s) as `./run-child_process-s2s.sh <W> <T>`, to perform the source-to-sink analysis.
After the script has finished, start `./run-child_process-crashes.sh`, optionally with a number of workers (default 5) and test timeout (default 20s) as `./run-child_process-crashes.sh <W> <T>`, to perform the unexpected-termination analysis.
Results: The output at the end of the former script constitutes part of the aggregate analysis numbers from Section 5.1 of the paper except limited to the `child_process` API (expect ~70,000 sinks reached with ~1,500 unique sink-source pairs before filtering, and ~40 unique sink-source pairs after filtering). It produces the SARIF files for manual review in a folder named `node/fuzzing/X-YYYY-MM-DD-HH-MM-SS`.
The output at the end of the latter script constitutes the remaining part of the aggregate analysis numbers from Section 5.1 of the paper except limited to the `child_process` API (expect 2 gadget candidates out of ~111,000 crashes).

(E3): Comparison on Node.js v21.0.0 GHUNTER, 10 human-minutes + <2 compute-hour + 50GB disk: The GHUNTER part of the comparison between GHUNTER and Silent Spring on Node.js v21.0.0.
Set-up: Follow (**S2**).
Preparation: Not applicable.
Execution: Start `./run-compare-ss-21.sh` to run the source-to-sink analysis for the relevant APIs for the comparison.
Results: This will output the results and also store them in the `node/fuzzing.ss21` folder. There will be 9 folders following the `X-YYYY-MM-DD-HH-MM-SS` naming scheme. Each maps to a row from Table 3 of the paper according to the mapping found in the project's README and contains two relevant files: `count.txt` for the number presented as "GC" in Table 3 and `compare.json` with the properties and corresponding sinks of each gadget candidate (validating them is a manual process). False negatives are derived as $FN = GT - TP$.

(E4): Comparison on Node.js v21.0.0 Silent Spring, 10 human-minutes + <5 compute-hours + 2GB disk: The Silent Spring part of the comparison between GHUNTER and Silent Spring on Node.js v21.0.0.
Set-up: Follow (**S3**) and checkout `14966b5`.
Preparation: Run `node --version` and ensure you are using v21.0.0.
Execution: Start `./compare.sh`.
Results: The script writes the raw results for the comparison as a folder per row of Table 3 in the paper in the `raw-data` folder. Each folder contains the raw output from Silent Spring as well as a `ghunter.log` file with the data for comparison. In particular, the last line (starting with `Candidates`) is the "GC" number from Table 3 and the data preceding it (starting from `all props`) contains the true and false positives data (validating them is a manual process). False negatives are derived as $FN = GT - TP$.

(E3): Comparison on Node.js v16.13.1 GHUNTER, 10 human-minutes + <2 compute-hour + 50GB disk: The GHUNTER part of the comparison between GHUNTER and Silent Spring on Node.js v16.13.1.
Set-up: Follow (**S2**).
Preparation: Not applicable.
Execution: Start `./run-compare-ss-16.sh` to run the source-to-sink analysis for the relevant APIs for the comparison.
Results: This will output the results and also store them in the `node/fuzzing.ss16` folder. There will be 11 folders following the `X-YYYY-MM-DD-HH-MM-SS` naming scheme. Each maps to a row from Table 2 of the paper according to the mapping found in the project's README and contains two relevant files: `count.txt` for the number presented as "GC" in Table 2 and `compare.json` with the properties and corresponding sinks of each gadget candidate (validating them is a manual process). False negatives are derived as $FN = GT - TP$.

[3]The full analysis can be performed by substituting "child_process" for "all" in the experiment steps, but this requires hardware similar to that described in the paper rather than hardware similar to that described in this appendix and is expected to take over 96 hours to complete.

(E6): Comparison on Node.js v16.13.1 Silent Spring, 10 human-minutes + <6 compute-hours + 2GB disk: The Silent Spring part of the comparison between GHUNTER and Silent Spring on Node.js v16.13.1.
Set-up: Follow (**S3**) and checkout `a6ae944`.
Preparation: Run `node --version` and ensure you are using v16.13.1.
Execution: Start `./compare.sh`.
Results: The script writes the raw results for the comparison as a folder per row of Table 2 in the paper in the `raw-data` folder. Each folder contains the raw output from Silent Spring as well as a `ghunter.log` file with the data for comparison. In particular, the last line (starting with `Candidates`) is the "GC" number from Table 2 and the data preceding it (starting from `all props`) contains the true and false positives data (validating them is a manual process). False negatives are derived as $FN = GT - TP$.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.