Mining Node.js Vulnerabilities via Object Dependence Graph and Query

Song Li and Mingqing Kang, *Johns Hopkins University;*
Jianwei Hou, *Johns Hopkins University/Renmin University of China;*
Yinzhi Cao, *Johns Hopkins University*

https://www.usenix.org/conference/usenixsecurity22/presentation/li-song

This artifact appendix is included in the Artifact Appendices to the Proceedings of the 31st USENIX Security Symposium and appends to the paper of the same name that appears in the Proceedings of the 31st USENIX Security Symposium.

August 10–12, 2022 • Boston, MA, USA

978-1-939133-31-1

A Artifact Appendix

A.1 Abstract

In the paper, we propose flow- and context-sensitive static analysis with hybrid branch-sensitivity and points-to information to generate a novel graph structure, called Object Dependence Graph (ODG), using abstract interpretation. ODG represents JavaScript objects as nodes and their relations with Abstract Syntax Tree (AST) as edges, and accepts graph queries—especially on object lookups and definitions—for detecting Node.js vulnerabilities.

We implemented an open-source prototype system, called ODGEN, to generate ODG for Node.js programs via abstract interpretation and detect vulnerabilities. Our evaluation of recent Node.js vulnerabilities shows that ODG together with AST and Control Flow Graph (CFG) is capable of modeling 13 out of 16 vulnerability types. We applied ODGEN to detect six types of vulnerabilities using graph queries: ODGEN correctly reported 180 zero-day vulnerabilities, among which we have received 70 Common Vulnerabilities and Exposures (CVE) identifiers so far.

In this artifact evaluation, we claim that OPGEN is capable of detecting all six types of vulnerabilities and found all the zero-day vulnerabilities.

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

- **Algorithm:** Mining Node.js Vulnerabilities via Object Dependence Graph and Query
- **Data set:** We use the self-generated dataset and it is included in the docker image
- **Run-time environment:** Ubuntu 20.04 is recommanded and tested. The main software dependencies are Python 3.7+, pip, npm, and Node.js 12+
- **Run-time state:** No
- **Metrics:** Number of detected vulnerable packages
- **Output:** The testing results are located in the "logs" folder of the running directory. All the detected vulnerable packages will be output to the "succ.log" file; All the un-detected packages will be output to the "results.log" file. You can get the number of the successfully detected packages by running "cat ./logs/succ.log | wc -l", during or after the running process.
- **Experiments:** You can download and load the docker, or set up the environment from the source code. Then run the pre-written scripts and see the results.
- **How much disk space required (approximately)?:** 10GB
- **How much time is needed to prepare workflow (approximately)?:** 10 to 30 mins
- **How much time is needed to complete experiments (approximately)?:** 200 mins
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** GPL v3.0
- **Data licenses (if publicly available)?:** GPL v3.0
- **Archived (provide DOI)?:**

A.3 Description

A.3.1 How to access

We provide two methods for testing, loading the docker image is highly recommanded:

- A docker image

  We uploaded our docker to Docker Hub. You can pull it by running

  docker pull iamthesong/odgen:latest

  Then you can attach to this docker by running

  docker run -it iamthesong/odgen bash

  After loading it, you should be able to see the environment

- A repository for the source code

  If you are not able to access the virtual machine and can not load the docker image, you can also try to clone our source code from the GitHub repository https://github.com/Song-Li/ODGen/tree/24d68fa810cae8c028cf36f269461e178c198c98 (commit hash: 24d68fa810cae8c028cf36f269461e178c198c98) and follow the instructions in the README.md to set up the environment.

A.3.2 Hardware dependencies

Recommended

- CPU: 16 cores
- Memory: 16GB

Minimum

- CPU: 4 cores
- Memory: 4GB

A.3.3 Software dependencies

If you want to start with the source code, Ubuntu 20.04 is recommended. This artifact requires Python 3.7+, pip, npm, and Node.js 12+.

A.4 Installation

A.4.1 Docker image

We prepared a docker image on Docker Hub. You can follow the commands mentioned in section A.3.1 to download the load the docker.

A.4.2 Source code

Setup the Environment If you want to start with the source code, we recommend you to use Ubuntu 20.04., you can simply cd into the source code folder and install the software dependencies by running:

./ubuntu_setup.sh

After the packages are successfully installed, you can setup the environment by running:

./install.sh

The script **install.sh** will install a list of required Python and Nodejs dependencies. Once finished, the environment is setup and we are ready to go.

Verify the Installation You can run the script **odgen_test.py** to verify the installation. The command is:

python3 ./odgen_test.py

If the environment is successfully set up, you should be able to see the tests are finished without errors. The end of the outputs should be like:

Ran 3 tests in XXXs

OK

A.5 Experiment workflow

As we claimed in the Abstract section and the Contributions part of section 1 in our paper, our main claim that needed to be evaluated is we found 43 application-level and 137 package-level zero-day vulnerabilities. Our tool can successfully found the vulnerabilities of those packages. We prepared the dataset and the related scripts to run our tool on top of the packages.

Besides the main claim, other evaluation results, including the performance, the code coverage, and the false-negative rate of our paper are also reproducible and reproduced by the reviewers. I will also include the steps and datasets to reproduce the related evaluation results in the next section.

A.5.1 File organization of our Docker Image

Once you log into the docker, all the files and folders are organized as follows:

```
.
|--projs: the source code and libs of our tool.
|--packages: all the zero-day vulnerable packages detected by our tool.
  |--code_exec: packages with zero-day arbitrary code execution vulnerabilities
    |--XX: package-name@version
      |--cve.txt: if it exists, it indicates the CVE identifier
      |--run.sh: a script to detect the zero-day vulnerability
  |--ipt: packages with zero-day internal property tampering vulnerabilities
  |--os_command: packages with zero-day OS command injection vulnerabilities
  |--path_traversal: packages with zero-day path traversal vulnerabilities
  |--proto_pollution: packages with zero-day prototype pollution vulnerabilities
  |--xss: packages with zero-day XSS vulnerabilities
|--examples: a few simple vulnerable examples
  |--pp_example.js: the prototype pollution example
  |--run_proto_pollution.sh: detect prototype pollution of pp_example.js
  |--motivating_example.js: the motivating example mentioned in the paper
  |--run_ipt.sh: detect internal property tampering of motivating_example.js
  |--run_os_command.sh: detect taint-style vulnerability of motivating_example.js
  |--clean.sh: clean up log files
|--back_up: recovery files (do not touch)
```

A.5.2 Dataset

Dataset 1: Zero-day vulnerable packages

- **dataset:** The 174 zero-day vulnerable packages that found by our tool. (Note that after our reporting, there are eight packages that are unpublished from NPM. Currently, we only have source code for 173 packages + one package, which is unpublished but cached on our server.)
- **location:** ∼/packages
- the CVEs they got: ∼/packages/xx/package-name@version/cve.txt (if exists)
- a script that runs the analysis on each of these folders/projects and detects the vulnerabilities: ∼/packages/xx/package-name@version/run.sh

  where xx = code_exec, ipt, os_command, path_trasversal, proto_pollution, and xss.

Note that considering the large size of the dataset, we are not able to upload the dataset to the GitHub repository. We uploaded the zipped dataset to Google Drive and if you are testing it by the source code, please download it, unzip it, and put it in the root directory of your machine.

Dataset 2: Legacy vulnerable packages

- **dataset:**The legacy vulnerable packages dataset mentioned in Section 6.3 of the paper, including 75 command injection vulnerable packages, 31 code execution vulnerable packages, 52 prototype pollution vulnerable packages, 87 path traversal vulnerable packages, and 11 internal property tampering (IPT) vulnerable packages.
- **location:** We uploaded it as a zip file to the GitHub Repo (https://github.com/Song-Li/legacy_benchmark)

Dataset 3: Randomly selected packages

- **dataset:** The 500 randomly selected packages from the NPM database.
- **location:** We uploaded it as a zip file to the GitHub Repo (https://github.com/Song-Li/random_500_npm.git)

A.5.3 Play with the examples

In the ∼**/examples** folder of the Docker image, we have a few simple vulnerable examples for you to get familiar with our tool. You can try the **run_ipt.sh, run_os_command.sh** or **run_proto_pollution.sh** to run our tool on top of the pp_example.js(a prototype pollution) example and the motivating_example.js(the motivating example introduced in our paper). You can also write your modules, use a similar command and test them out.

A.6 Evaluation and expected results

A.6.1 Evaluation

Zero-day vulnerable packages detection (Dataset 1) Totally we have six different types of vulnerabilities, they are command injection, code execution, prototype pollution, path traversal, cross-site scripting, and internal property tampering. Each of them can be tested by running a command in the root directory of the source code:

- Command injection: ./scripts/os_command.sh
- Code execution: ./scripts/code_exec.sh
- Prototype pollution: ./scripts/prototype_pollution.sh
- Path traversal: ./scripts/path_traversal.sh
- Cross-site scripting: ./scripts/xss.sh
- Internal property tampering: ./scripts/ipt.sh

To reproduce the results, you can pick a vulnerability type and run the corresponding script.

Note that the scripts will try to run our tool parallelly, so you will not see the progress. Once you run a script, you should be able to see a message that says "new instance". You can check how many processes are still running by the command: *screen -ls*. You can also attach to a specific process by running: *screen -r XXX*(XXX means the name of the screen). Once all the processes are finished, you can check the result and run another script.

The testing results are located in the *logs* folder of the running directory. All the detected vulnerable packages will be output to the *succ.log* file; All the un-detected packages will be output to the *results.log* file. You can get the number of the successfully detected packages by running *cat ./logs/succ.log | wc -l*, during or after the running process.

If you finished checking one vulnerability type, please run *./clean.sh* to remove the logs and temporary files before checking another one.

Note that since the order of the testing functions is randomized, you may encounter some un-detected packages. For the un-detected packages, you may run them independently follow the instructions in README.md, or, go to *∼/packages/vulneralbility-type/package-name@version/* and run the *run.sh*

False negative rate (Dataset 2) The false-negative rate is introduced in Table 9 of the paper, which is measured on top of the legacy vulnerable packages. The steps to reproduce it is:

- Login to our Docker by the command *docker run -it iamthesong/odgen bash*
- Make sure you are in the root directory of the docker, and download the dataset by *git clone https://github.com/Song-Li/legacy_benchmark.git*
- Go into the downloaded dataset by *cd legacy_benchmark/* and unzip the dataset by *unzip legacy_benchmark.zip*
- Go into the source code directory by *cd /projs/ODGen/*. Test a type of vulnerability by *./odgen.py -t VUL_TYPE --list /root/legacy_benchmark/VUL_TYPE.list -aq --nodejs --timeout 120 --parallel 16*

Note that

- There are two locations in the last command that use *VUL_TYPE*. *VUL_TYPE* should be replaced by *os_command, ipt, proto_pollution, path_traversal* or *code_exec*
- The *--parallel* argument is used to run ODGen parallelly. In my case, I use 16 to indicate that I want to run 16 processes together. You can adjust the argument based on the number of CPU cores of your device
- The *--timeout* argument is used to set the timeout of a single test. We recommend 300 to make sure it works. In most cases, 120 should be enough.

If the number is less than expected, we need to run multiple times on those packages. You can simply run the same command multiple times (without cleaning the logs), and the results will be logged to the */root/projs/ODGen/logs/succ.log* file, cumulatively. To remove the duplicates, you can go into the */root/projs/ODGen/logs/* folder and run *awk '!x[$0]++' succ.log > outfile.succ*. The generated file *outfile.succ* will be the detected list.

Code coverage (Dataset 3) The code coverage rate is introduced in Figure 9 of the paper, which is measured on top of the 500 randomly selected Node.js packages. We prepared the statement-level code coverage API and the randomly selected 500 packages for testing.

Steps to reproduce:

Table 1: Expected Detection Results for Zero-day Vulnerable Packages

	Command Injection	Code Execution	Prototype Pollution	Path Traversal	Cross-site Scripting	IPT
#Packages	80	14	19	30	13	24
#Unpublished	2	4	0	0	0	0
#Expected	76∼78	9∼10	17∼19	30	12∼13	23∼24

Table 2: Expected True Positive Packages on Legacy Vulnerable Packages

	Command Injection	Code Execution	Prototype Pollution	Path Traversal	IPT	Total
#Packages	75	31	52	87	11	256
#Claimed True Positive	67	20	40	55	7	189
#Expected True Positive	67	20∼21	39∼40	55∼ 56	7∼ 10	189∼194

Table 3: Reproduced Code Coverage Rate

Code Coverage	Percentage of Packages
0% to 10%	5.52%
10% to 20%	5.25%
20% to 30%	8.01%
30% to 40%	2.76%
40% to 50%	2.76%
50% to 60%	6.63%
60% to 70%	2.76%
70% to 80%	6.35%
80% to 90%	11.60%
90% to 100%	48.34%

- Login to our Docker by the command *docker run -it iamthesong/odgen bash*
- Make sure you are in the root directory of the docker, and download the dataset by *git clone https://github.com/Song-Li/random_500_npm.git*
- Go into the downloaded dataset by *cd random_500_npm/* and unzip the dataset by *unzip ./random_500.zip*
- Go into the source code directory by *cd /projs/ODGen/*.
- Update the source code to the latest version by *git pull*
- Start the testing by running *./odgen.py -t os_command -ma –list /random_500_npm/random_500.list --timeout 30 --parallel 20*
- During the running process, you can go to the tools folder by *cd /root/projs/ODGen/tools* and check the results on the fly by running *python get_code_coverage_dis.py*. This script will output the results directly. You can run this command multiple times to see how the code coverage changes during the evaluation.

You can check how many processes are running by *screen -ls*. If all processes are finished, you can check the final result. Note that the code coverage raw data is logged in *ODGen/logs/stat.log*. You can also take a look if you want!

Note that not all of the packages will report code coverage, There are two reasons for that:

- Since the packages are randomly selected, there are many packages that do not meet the requirement of the NPM standard. For example, some of them do not have an entrance file, some of them do not include a package.json file, and some of them are demo packages without any meaningful content. For those packages, ODGen will not report the code coverage;
- It is possibly happening for packages running into a timeout. ODGen will not output the code coverage results for timeout packages since those results can not reflect the real code coverage of ODGen.

A.6.2 Expected results

Zero-day vulnerable packages detection The number of all the packages, unpublished packages, expected detected packages and the estimated running time are listed in Table 1

False negative rate The number of all the packages, claimed true positive packages, expected detected packages are listed in Table 2

Code coverage The distribution of the code coverage should be comparable to Figure 9 of the paper. The results that reproduced by the reviewers are listed in Table 3

A.7 Troubleshooting

Zero-day vulnerable packages detection If you can not get the expected results, you can try to restart the docker and see if it can run smoothly without the influence of the cache.

False negative rate If you can not get the expected results, you can try to:

- When you run the tool multiple times, try to change the number of *--parallel* each time. For example, we can use *--parallel 17* for the first time, and *--parallel 19* for the second time. In that way, each process may start from different packages and it may be faster to generate the results.
- Since the number of packages with code execution vulnerability is not very large. If your device has enough computing resources, for example, more than 20 CPU cores. You can try to set the *--parallel* argument to *--parallel 31* to make sure every vulnerable package can use an independent process. After doing this, you can check how many packages are still running by using *screen -ls*.

A.8 Experiment customization

You are very welcome to test our tool on top of your customized packages. To do so, please go to the ∼/example folder and write your package follow the NPM package standard, or write a module like the ∼/example/pp_example.js and the ∼/example/motivating_example.js.

Once you prepared the module, you can check out the README.md file in the source code repository and follow the instructions to run the corresponding commands.