SandDriller: A Fully-Automated Approach for Testing Language-Based JavaScript Sandboxes

Abdullah AlHamdan and Cristian-Alexandru Staicu,
CISPA Helmholtz Center for Information Security

https://www.usenix.org/conference/usenixsecurity23/presentation/alhamdan

This artifact appendix is included in the Artifact Appendices to the Proceedings of the 32nd USENIX Security Symposium and appends to the paper of the same name that appears in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Open access to the Artifact Appendices to the Proceedings of the 32nd USENIX Security Symposium is sponsored by USENIX.

USENIX'23 Artifact Appendix: SANDDRILLER: A Fully-Automated Approach for Testing Language-Based JavaScript Sandboxes

Abdullah AlHamdan, Cristian-Alexandru Staicu
CISPA Helmholtz Center for Information Security
{abdullah.alhamdan, staicu}@cispa.de

A Artifact Appendix

A.1 Abstract

In this artifact, we showcase SANDDRILLER, the first tool for testing language-based sandboxes JavaScript, supporting both client-side and server-side sandboxes. SANDDRILLER takes as input a set of self-contained JavaScript files (corpus), interposes oracles using instrumentation, and executes each instrumented test case inside a target sandbox. Additionally, SANDDRILLER also recombines ingredients from known exploits to generate variants of the files in the corpus that are more likely to trigger a bug in the target sandboxes. We present experiments that demonstrate how SANDDRILLER works overall (single input file, variant generator) and partially replicate important results from the paper (V8 tests corpus with the `vm2` sandbox). SANDDRILLER is publicly available at https://github.com/vdata1/SandDriller. For this artifact evaluation, we seek the following badges: available and functional.

A.2 Description & Requirements

In this section, we describe the software and hardware requirements for running SANDDRILLER and explain how to obtain the corpora used in the evaluation of the paper.

A.2.1 Security, privacy, and ethical concerns

There are no risks for the reviewers of this artifact with respect to security and privacy of their machines. SANDDRILLER identified 12 zero-day vulnerabilities in widely-used, open-source sandboxes. A security advisory was published for each of these findings.

A.2.2 How to access

The source code is available on our GitHub repository https://github.com/vdata1/SandDriller/releases/tag/1.0, including an important README file that describe in detail the usage of the tool and the different configuration options available.

A.2.3 Hardware dependencies

For the paper's evaluation, we ran SANDDRILLER on a server with 64 Intel Xeon E5-4650L@2.60GHz CPU cores and 768GB of memory. However, SANDDRILLER does not require any specific hardware feature, so it can run on any other machine running a Linux distribution. Nonetheless, since SANDDRILLER makes extensive use of multi-threading during testing, we recommend that the number of parallel workers be set to maximum the number of physical threads available on the machine. In Section A.4.2, we explain how to set this important configuration option.

A.2.4 Software dependencies

While SANDDRILLER might run on other operating systems, we require a Linux distribution for the evaluation. For obtaining the results described in this artifact, we require running the tool using Node.js version 14.15. We recommend using the Node Version Manager (nvm) [1] to install this exact version of Node.js. We also expect the machine to have `git` and `npm` installed and correctly configured. To showcase the tool's capability we run tests using two sandboxes: `vm2` and `safe-eval`. Both these tools are declared as third-party dependency in the `package.json` file, so they do not need to be installed separately. However, since we do not ship them with our tool, the input corpora need to be downloaded and configured separately, as described below.

A.2.5 Benchmarks

To test the sandboxes, we used ECMAScript Conformance Test Suite and V8 engine's test suites as benchmarks. SANDDRILLER takes each of these tests, instrument them, and run them inside the sandbox under test. In our experiments, we used ECMAScript test cases available at https://github.com/tc39/test262/tree/99b2a70789b27d433f9036b98572a4443d91e01f/test, and V8 test cases available at https://github.com/nodejs/node/tree/e46c680bf2b211bbd52cf959ca17ee98c7f657f5/deps/

[1] https://github.com/nvm-sh/nvm/

v8/test/mjsunit. As described below, during installation, these repositories need to be cloned into a specific subfolder.

A.3 Set-up

A.3.1 Installation

First, clone the SANDDRILLER repository locally in a directory we will call $PATH_TO_SANDDRILLER. To install the required third-party packages, run in the project's main directory npm -prefix . install .. This will install all the required dependencies and all the npm-available (vulnerable) sandboxes used in our experiments.

Next, clone the V8 corpus inside the $PATH_TO_SANDDRILLER/Dataset/ folder:

```
cd Dataset
git clone https://github.com/nodejs/node/
git reset --hard e46c680
cd ..
```

A.3.2 Basic Test

To run a simple test with SANDDRILLER, go to test directory by typing on the command line cd $PATH_TO_SANDDRILLER/test and run node run-multi-proc.js.
As a result, RESULTS.csv will be written on $PATH_TO_SANDDRILLER/Results/ showing the test results of SANDDRILLER. Results will also be shown on the terminal, i.e., a JSON result object for each test, followed by a summary of the results.

When executing SANDDRILLER on a fresh installation, it uses a toy corpus containing three JavaScript files. SANDDRILLER should report two security violations and a crash for this corpus.

A.4 Evaluation workflow

A.4.1 Major Claims

SANDDRILLER is a testing approach for automatically detecting sandbox escape vulnerabilities in real-world JavaScript sandboxes. To successfully isolate untrusted code, JavaScript sandboxes must block access to foreign references and prevent the side effects of hosting third-party code during runtime, such as getting stuck in an endless loop. SANDDRILLER aims to automatically synthesize exploits that violate this objective by escaping the sandbox. It first interposes checks that at execution time detect foreign references pointing outside of the sandbox, and subsequently exploits such references, creating an end-to-end exploit.

We make the following claims about our prototype:

(C1): *Starting with (a set of) benign JavaScript file(s) as input,* SANDDRILLER *can detect problematic references at runtime, which can be used to escape the sandbox. Concretely,* SANDDRILLER *synthesizes exploits that attempt to access privileged operations outside the sandbox, and/or test write values into the global scope, outside the sandbox. We have conducted experiment (E1) and (E3) as described in Section A.4.2 to demonstrate this capability.*

(C2): SANDDRILLER *can construct exploits by synthesizing variants of the input files. By using a variant generator it provides more comprehensive tests for verifying the security of a sandbox. We have demonstrated this through experiment (E2) as described in Section A.4.2.*

These claims collectively support our paper's main assertion that SANDDRILLER is an effective testing approach for automatically detecting zero-day sandbox escape vulnerabilities.

A.4.2 Experiments

Since replicating our entire testing campaign would incur a significant effort on the reviewers' side, we show how SANDDRILLER works with a single file as input and with a fairly large corpus (5,000+ JavaScript files from the V8 tests), for both using a single sandbox and a single Node.js version.

(E1): *[First exploit] [For one valid test case: 15 human-minutes + 5 compute-minutes]:* In this experiment, we show step-by-step how to come up with the first working exploit using vm2 sandbox and a single test case.

How to: To successfully accomplish this experiment, we start with configuring and running our tool, then run the instrumented code on the sandbox, and finally, apply delta debugging to reach the minimal working code for the exploit.

Preparation: First, we start editing the source code of the tool for the corpus for this experiment by uncommenting line number **138** in test/run-multi-proc.js (set regress-746909.js as the single file in the corpus). Moreover, go to test/process-runner.js and make vm2 as the sandbox to test by editing line number **10** to const sandbox = "vm2";.

Execution: Run SANDDRILLER as mentioned in A.3.2. As a result, SANDDRILLER will write the successful instrumented test case on path /tmp/res/. First, let us inspect the content of the generated file regress-746909.js, which is an instrumented version of the file in our corpus https://github.com/nodejs/node/blob/main/deps/v8/test/mjsunit/regress/regress-746909.js. It contains code for verifying potential foreign references and for attempting to escape the sandbox using such references. For example, for each function invocation, SANDDRILLER obtains the root prototype of its result let grtA = getRootPrototype(r); and attempts to modify the corresponding root prototype grtA.FIA = 'FI: Got it?';. Copy the exploit (instrumented

test case) and host it on the target sandbox, vm2. To this end, we provided template for each sandbox in `templates_sandboxes` for time-saving. Paste the generated exploit in `exploit.js` and run the `vm2.js` file using Node.js. Observe how the global root prototype changes as a result of running the code inside the sandbox, i.e., a new property is added on the root prototype. Optionally, apply manual delta debugging by deleting the unnecessary lines of code in the exploit to get the minimal working exploit. To that end, after any removing spurious lines in the code, rerun the exploit to verify that the desired side-effect in the global scope is still present. To get the minimal working exploit, a sequence of lines of code deletion and reruning of the code might be required. For this specific test case, we recommend deleting the first **122, 139-148, 150-180, 180-192, 204-end of the code** lines to get a working exploit. The code at this stage requires more editing to produce the final working exploit showed in Figure 1 of the paper. However, we provided a working exploit for this experiment, showing the possible steps performed during delta debugging in `demo_results/E1.js`.

Results: After completing the experiment, the detailed result can be found in `Results/RESULTS.csv`

(E2): *[Use the Variant Generator] [For one valid test case 20 human-minutes + 5 computing-minutes]:*

How to: To successfully accomplish this experiment, we start with configuring our tool to **enable** the generator, then run the instrumented code on the sandbox, and finally, apply delta debugging to reach the minimal working code for the exploit.

Preparation: First we enable the generator in `test/process-runner.js` by assigning `const useGenerator = true` at line **11** and using `safe-eval` as the sandbox to test. Then, uncomment line number **`141`** in `test/run-multi-proc.js`, and comment line 138.

Execution: Run SANDDRILLER as mentioned in A.3.2. As a result, SANDDRILLER will write variants of successful instrumented test cases with on path `/tmp/res/`. Copy the instrumented test case, named "`array-push2_v1.js`" and host it on the target sandbox, `safe-eval`. As before, we provided a template for each sandbox in `templates_sandboxes` for time-saving: paste the explit in the `exploit.js` file and run `safe-eval.js` with Node.js to observe its side effect in the global scope (a new property on the root prototype). We Optionally, apply delta debugging by deleting the unnecessary lines of code in the exploit to get the minimal working exploit. For this specific test case, we recommend deleting **3-123, 127-157, 129-162, 165-196, 199-261, 266-284, 289-422, 426-514** lines to get working exploit. The code at this stage requires more editing to come up with the final working exploit. However, we provided a working exploit for this experiment, showing a possible result of delta debugging in `demo_results/E2.js`. We draw the reader's attention to the code `throw function thrower() {...}`, which is a code fragment that was not part of the original file in the corpus, but was injected by the variant generator.

Results: After completing the experiment, the statistical result can be found in `Results/RESULTS.csv`

(E3): *[Running* SANDDRILLER *on a benchmark] [5 human-minutes + 1 computing-hour]:*

How to: To successfully accomplish this experiment we run SANDDRILLER on a fairly large corpus consisting of 5,000+ V8 tests. For simplicity, the variant generator is disabled.

Preparation: First make sure to delete all generated test cases on `/tmp/res` and delete the results file to work in a clean environment. Then, uncomment line number **`144`** in `test/run-multi-proc.js` and change the number of threads to 16 at line 12 by `const POOL_SIZE = 16;`. Make sure the generator is disabled by setting `const useGenerator = false;`. The sandbox from the previous experiment should remain the same for this experiment.

Execution: Run SANDDRILLER as mentioned in A.3.2.

Results: After completing the experiment, the detailed results can be found in `Results/RESULTS.csv`, and a summary of the the experiment will be printed in the terminal. The results list all the tests in which SANDDRILLER succeeded to break the sandbox, test cases that cause a hard crash of the sandbox, and more details such as execution time or number of oracle checks performed at runtime. Feel free to attempt hosting any of the produced exploits inside the vulnerable sandbox (`safe-eval`) and observe its side effect in the global scope, as before.

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/.