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

Abdullah AlHamdan
CISPA Helmholtz Center for Information Security
abdullah.alhamdan@cispa.de

Cristian-Alexandru Staicu
CISPA Helmholtz Center for Information Security
staicu@cispa.de

Abstract

Language-based isolation offers a cheap way to restrict the privileges of untrusted code. Previous work proposes a plethora of such techniques for isolating JavaScript code on the client-side, enabling the creation of web mashups. While these solutions are mostly out of fashion among practitioners, there is a growing trend to use analogous techniques for JavaScript code running outside of the browser, e.g., for protecting against supply chain attacks on the server-side. Irrespective of the use case, bugs in the implementation of language-based isolation can have devastating consequences. Hence, we propose SANDDRILLER, the first dynamic analysis-based approach for detecting sandbox escape vulnerabilities. Our core insight is to design testing oracles based on two main objectives of language-based sandboxes: Prevent writes outside the sandbox and restrict access to privileged operations. Using instrumentation, we interpose oracle checks on all the references exchanged between the host and the guest code to detect *foreign references* that allow the guest code to escape the sandbox. If at run time, a foreign reference is detected by an oracle, SANDDRILLER proceeds to synthesize an exploit for it. We apply our approach to six sandbox systems and find eight unique zero-day sandbox breakout vulnerabilities and two crashes. We believe that SANDDRILLER can be integrated in the development process of sandboxes to detect security vulnerabilities in the pre-release phase.

1 Introduction

Lightweight, language-based isolation for JavaScript was pioneered by systems like Google Caja [3] and Douglas Crockford's ADsafe [1]. They allow code from different parties to run in the same origin inside the browser, i.e., enabling the creation of web mashups. However, due to the large number of vulnerabilities in these systems and the increased adoption of safer alternatives, e.g., `iframe` isolation with exchange of post messages, these solutions are mostly abandoned today. However, the core ideas advocated by these systems continue to thrive in the modern JavaScript ecosystem.

```
1 const {VM} = require("vm2");
2 let sandbox = new VM();
3 let code = `
4  let res = eval("import('./foo.js');")
5  res.__proto__.__proto__.toString.constructor("
       return this")().process.mainModule.require("
       child_process").execSync("cat /etc/passwd");
6 `;
7 sandbox.run(code);
```

Figure 1: CVE-2021-23449, a critical sandbox breakout vulnerability found by SANDDRILLER. The exploit uses the result of an import call to escape the `vm2` sandbox and invoke arbitrary operating system commands.

Emerging JavaScript use cases demand cheap sandboxing that can run outside the browser. For example, Cloudflare [6] uses it for efficient resource sharing, Tripadvisor [9] for server-side rendering, Embark [4], and Agoric [8] for blockchain applications, Moddable [7] for the IoT domain, and Box.js [2] for malware analysis. Many popular JavaScript sandboxes have language-based techniques at their core: `vm2` - a package with millions of weekly downloads, Agoric's `realms-shim` and `ses` - two implementations of early-stage ECMAScript proposals, Salesforce's `near-membrane`, and `SandTrap` [14] - a sandbox for trigger-action IoT platforms. To understand the threat model of these sandboxes, we first perform an empirical study of their features and the assumptions they make about the code they execute.

We find that many existing solutions use Node.js' `vm` module for base isolation and interpose additional security checks through the `Proxy API`. Hence, they are incompatible with older sandbox scrutiny techniques proposed before the adoption of these APIs, e.g., Politz et al. [54].

Let us consider the example in Figure 1 to illustrate why this is the case. The untrusted code depicted in lines 4-5 escapes the sandbox by traversing the prototype chain to the root prototype and invoking the `Function` constructor belonging to the host code. This, in turn, gives access to the

https://t.me/learningnets

highly-privileged `require` function, which then allows arbitrary command execution. The root cause of the problem is that the reference assigned to `res` in line 4 has a root prototype that points outside of the sandbox. We call these references *foreign references* and we observe that most of the sandboxes we consider aim to prevent at all costs that such references reach inside the sandbox. For the presented example, the maintainers of `vm2` and `SandTrap`, two sandboxes affected by this vulnerability, fixed their implementations in response to our reports by disallowing `import` calls inside the sandbox. These calls were recently introduced in the ECMAScript 2015 standard, and their usage triggers the creation of a foreign reference inside Node.js' `vm` module, a building block of the aforementioned sandboxes.

The example above shows that security problems appear when obscure, newly-introduced language features are used inside real-world sandboxes. Hence, for state-of-the-art approaches like that of Politz et al. [54] to find such bugs, one would need to consider the semantics of the entire JavaScript language and its custom extensions introduced by Node.js. We note that such an endeavor is beyond the capabilities of existing static analyses for JavaScript. Moreover, the interposition through `Proxy API` used by modern sandboxes further complicates the implementation of static analyses. Thus, we propose a radically different way of hardening sandboxes based on dynamic analysis.

Our idea is to draw inspiration from recent advances in engine fuzzing [15, 30, 31, 37, 50, 53, 70] and apply it to sandbox testing. However, prior work in JavaScript fuzzing mostly uses naïve crash oracles that cannot capture subtle bugs in the containment logic of language-based sandboxes. For testing engines written in memory unsafe languages like C++, this coarse grain approximation suffices to identify potential memory corruption vulnerabilities. For our case, though, since we expect most of the bugs to be in the JavaScript code, a crash oracle is not effective at identifying security violations. Thus, we propose novel oracles that identify foreign references by traversing the prototype chain.

SANDDRILLER is a black-box testing approach for language-based sandboxes. It accepts a corpus of JavaScript programs as input, and using instrumentation, it deploys relevant oracle checks that search for foreign references. If a problematic reference is identified, SANDDRILLER validates the candidate security problem by attempting to write outside the sandbox or to invoke privileged operations using the suspicious reference. If this succeeds, the instrumented program is minimized using delta debugging, producing a condensed proof-of-concept sandbox breakout. SANDDRILLER also creates variants of the seed programs by introducing ingredients commonly seen in known sandbox breakout exploits.

We evaluate our approach using six sandboxes, three different Node.js versions, and two sources of seed programs: The ECMAScript conformance test, also known as Test262, and V8's unit tests. In total, we run SANDDRILLER for 17.27 hours, performing more than three billion oracle checks. We observe 115,085 security violations and 48 hard crashes, affecting five out of the six considered sandboxes. However, most of these issues are caused by a handful of root causes, hence, after manual investigation and grouping, we report 13 distinct security problems in the analyzed sandboxes. At the time of writing, eight of them were fixed, and one sandbox was marked as deprecated in response to our reports.

In summary, our contributions are the following:

- We present a fully automated approach for testing JavaScript sandboxes. To the best of our knowledge, ours is the first dynamic analysis-based approach for hardening language-based sandboxes.
- We present extensive empirical results: first, a study of real-world JavaScript sandboxes and their objectives and second, an evaluation of our automated testing approach on six sandboxes, using 46,606 seed programs.
- We identify 12 confirmed zero-day security issues in open-source projects. Most of them are sandbox breakout vulnerabilities accompanied by a working exploit. We were assigned eight security advisories for our findings, most of them evaluated as having "critical" severity.

2 Study of isolation solutions for JavaScript

Instead of specifying a priori a list of security objectives that JavaScript sandboxes must adhere to, we perform an empirical study of the existing solutions in the ecosystem: (i) we carefully read the documentation of the considered sandboxes, (ii) we study in detail the previously reported sandbox breakout vulnerabilities and their fixes, (iii) we set up the sandboxes locally and run simple test programs to understand their capabilities, (iv) we audit the sandboxes' source code to identify potential flaws. We focus on general-purpose sandboxes that claim to allow safe execution of untrusted code. That is, we test critical components of the software supply chain instead of full-fledged systems using them. We do so because sandboxes are often available as open-source software, they are easy to set up and test, e.g., no legal implications for attempting to escape a given sandbox locally, and securing sandboxes can positively impact multiple clients at once. Nonetheless, we observe that there are real systems that can potentially benefit from securing the sandboxes above: TripAdvisor and Screep use `isolated-vm`, Agoric and Lavamoat use `ses`, NodeRED and Embark use `vm2`, and Salesforce uses `near-membrane`.

In Table 1, we compile a list of language-based isolation techniques to be considered in the empirical study. While we have no way to ensure that this list is comprehensive, over the course of several months, we performed regular queries for the keywords "isolation", "sandbox", "cage", and "jail", both on

https://t.me/learningnets

npm and in the scientific literature to identify JavaScript sandboxes in use. In the second column of the table, we show the number of previously known security vulnerabilities for these systems, obtained by mining public vulnerability databases and by studying the issues in the sandboxes' repositories. Appendix A further lists all the vulnerabilities we consider in our study. We acknowledge that there are vast differences in the popularity, maintenance, and implementation rigor of the considered sandboxes. Nonetheless, we believe that this list effectively captures both the relevant industrial systems and the recent scientific ideas in this area.

As a result of our investigation, we compiled a list of four security objectives that the existing JavaScript language-based sandboxes aim to fulfill, which we present below.

2.1 Security objectives

The main use case for the considered sandboxes is enabling safe execution of dangerous JavaScript code. We call the trusted, first-party code, *host code*, and we assume it has all the capabilities offered by the underlying JavaScript platform. In contrast, the *untrusted* or *guest* code's capabilities need to be limited to prevent damage to the underlying system, e.g., removing important files from the disk.

We assume the guest code actively tries to bypass the security control that aims to restrict its privileges. That is, in this work, we assume an adversary with malicious intent. We also note that there is an inherent tension between powerful isolation and increased usability. Solutions that only allow data sharing through copies, e.g., web workers, are inherently more secure than the ones allowing shared references. However, we observe that multiple use cases require tight integration between host and untrusted code. For enabling these, many popular JavaScript sandboxes rely on lightweight isolation solutions. We now proceed to discuss the identified security objectives.

Security objective 1 - SO_1

The sandbox should prevent side effects in the runtime, e.g., writing of values in the global scope.

Builtin objects in the global scope called intrinsics in JavaScript can be modified at will by user code. Prototype pollution [39], a vulnerability that received a lot of attention recently, is also based on the code's ability to compromise the integrity of the runtime. It is known to lead to serious security problems, e.g., remote code execution [57]. Hence, sandboxes prevent such modifications by containing any change to the runtime inside the sandbox.

Let us consider the following example:

```
1 import Realm from 'realms-shim'
2 let sandbox = Realm.makeRootRealm();
3 sandbox.evaluate(`
4    delete JSON.parse;
5    // JSON.parse undefined inside the sandbox
6 `);
7 JSON.parse("{}"); // available outside
```

The untrusted code shown in lines 4-5 deletes the API for parsing JSON. This modification is immediately effective inside the sandbox, e.g., in line 5, but it does not affect the host code, thus, the API can be safely invoked in line 7.

Security objective 2 - SO_2

The sandbox should restrict access to privileged operations, e.g., to `require` API.

By default, all the considered sandboxes disable access to powerful Node.js-specific intrinsics, e.g., the `require` API for including additional code. However, users can specify a list of endowments, i.e., shared references that will be available inside the sandbox. Let us consider the example below where the powerful `process.kill()` method is endowed inside the sandbox with the name `kill`:

```
1 const ivm = require('isolated-vm');
2 let isolate = new ivm.Isolate();
3 let sandbox = isolate.createContextSync();
4 sandbox.global.setSync('kill', new ivm.Reference(
      process.kill));
5
6 sandbox.evalSync(`
7    // process.kill() not available in the sandbox
8    kill(0); // only through endowments
9 `);
```

Considering that users can share arbitrary references inside the sandbox, the security of the sandbox is highly dependent on the list of endowments.

Security objective 3 - SO_3

The sandbox should prevent blocking the main loop of the host process, e.g., by using infinite loops.

The Node.js runtime is a single-threaded environment that relies on fair usage of the event loop. Thus, if the untrusted code inside the sandbox attempts to run an endless computation, the sandbox should prevent that the host code is also blocked, ideally by providing a timeout functionality. Let us consider the following example using the `notevil` sandbox:

```
1 let sandbox = require('notevil')
2 try {
3     sandbox('while (true);')
4 } catch(e) {
5     // Infinite loop detected
6 }
```

The sandbox detects the endless computation performed by the untrusted code, and stops execution. Hence, the host code is not affected by this naive attack.

https://t.me/learningnets

Security objective 4 - SO_4

The sandbox should prevent crashing the host process, e.g., through memory exhaustion.

Hard crashes of the runtime are to be avoided at all costs in JavaScript. This has to do with the evolution of the language: in the browser, pieces of code from different origins should be allowed to run in the same runtime, but they should not be able to influence each other, e.g., crashing each other. Some of the sandboxes we considered attempt to prevent this as well. Let us consider an example using the `jailed` sandbox:

```
1 let sandbox = require('jailed');
2 new sandbox.DynamicPlugin("let a=[]; while(1) {a.
    push(new Array(4294967295))}");
3 setTimeout(() => {
4     console.log("Successfully executed");
5 }, 60000)
```

The untrusted code attempts to exhaust the allocated memory and hence, crash the runtime. The sandbox, however, ensures that after the memory is exhausted and the guest code crashes, the host can still execute the closure in lines 3-5.

We now proceed to see how two important classes of sandboxes address the security objectives introduced above.

2.2 Runtime-based sandboxes

The first category of sandboxes we identified use strong primitives provided by the runtime for isolating untrusted code. For example, `BreakApp` and `jailed` use processes for isolation, `TreeHouse` and `deno-vm` use web workers, and `isolated-vm` uses the `Isolate` interface provided by the JavaScript engine. At the start of our project, we were convinced that these tried and tested building blocks are robust enough to prevent most attacks. As seen in the upper part of Table 1, these sandboxes aim to address all four security objectives. Moreover, we did not identify publicly-available security advisories for any runtime-based sandboxes.

It is worth mentioning that we were surprised to see that these systems are not the main choice of developers and that the popular sandboxes we identified on npm, e.g., `vm2` and `ses`, were lighter language-based solutions. We believe this has to do with the performance penalty incurred by runtime-based systems and their inability to deal with complex shared state. Most of these sandboxes do not allow shared references, and advocate instead for a call-by-copy strategy, which may be too heavy for certain use cases, e.g., run arbitrary third-party libraries in a sandbox and interact with them.

Nonetheless, `isolated-vm` allows shared references between the host and the untrusted code. After studying the documentation and the source code, we noticed that any such reference allows arbitrary code execution in the host's context. The untrusted code in the example above that uses the `kill` method can be modified to write arbitrary files on the disk:

```
1 kill.getSync('constructor').applySync(undefined, [
    'process.mainModule.require(\"child_process\")
    .execSync(\"touch success.txt\")']).applySync(
    undefined, []);
```

Using the legitimate API provided by the sandbox, we request the `constructor` property of the shared reference and use it to inject arbitrary code in the host context. Hence, the untrusted code can access powerful intrinsics like `process` and `require`. We call this problem a *capability leak*, in which a granted endowment can be used for privileged escalation.

We reported this problem to the maintainers of `isolated-vm` and their initial response was that users of the library are aware of this known attack vector. After providing several vulnerable usages in the clients of `isolated-vm`, the problem was fixed, and a security advisory was issued for our bug report (CVE-2021-21413). To the best of our knowledge, this is the first vulnerability report published for a runtime-based JavaScript sandbox.

Another problem we discovered involves the `jailed` sandbox. Even though this sandbox runs untrusted code in a separate process, hence addressing objective SO_1, the new process fails to restrict access to the powerful builtins described earlier, allowing privilege escalation. After reporting this issue, CVE-2022-23923 was issued to warn about this danger.

These results show that while runtime-based sandboxes use powerful isolation primitives, they can still be vulnerable due to the way these primitives are used and exposed to users.

2.3 Language-based sandboxes

When considering lighter isolation alternatives in Table 1, we were surprised to see that most of them do not address SO_3 and SO_4. Even when they do, we identify ways to bypass the protection. For example, the protection on the maximum number of iterations in `notevil` can be side-stepped by traversing large arrays using higher-order functions.

Considering that two of the considered language-based sandboxes, i.e., `realms-shim` and `ses`, correspond to early-stage ECMAScript proposals, we contacted their authors to better understand their threat models. They confirmed that these two sandboxes do not consider SO_3 and SO_4 and stated that it is the responsibility of their users to build additional defenses against such attacks, e.g., by running each sandbox in a separate UNIX process.

To address SO_1, all the considered sandboxes attempt to hide the global scope using different strategies: Node.js' `vm` module (`SandTrap`, `realms-shim`, `near-membrane`, `safe-eval`), interpose proxies[1] using the `with` statement (`ses`), disallowing access to global variables during interpretation (`notevil`), or shadowing them using redeclaration (`MIR`). Additionally, `ses` also proposes freezing the intrinsics to prevent any attempts at tampering with the global objects.

[1] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy

https://t.me/learningnets

	Sandbox	Type	Vulns.	SO_1	SO_2	SO_3	SO_4	Sandboxing strategy	Downloads
Runtime-based	TreeHouse [32]	C	0 / 0	●	●	●	●	worker threads with post messages	n/a
	BreakApp [65]	S	0 / 0	●	●	●	●	OS processes with IPC	n/a
	jailed	C+S	0 / 0	◐	◐	●	●	OS processes with IPC	62
	deno-vm	S	0 / 0	●	●	●	●	worker threads with Deno	173
	isolated-vm	S	0 / 0	●	◐	●	●	expose `V8`'s `Isolate` API	12,097
Language-based	vm2	S	0 / 15	◐	◐	◐	○	`vm` module and membranes	3,547,348
	realms-shim	C+S	0 / 2	●	◐	○	○	`vm` module and membranes	405
	ses	C+S	0 / 0	●	●	○	○	membranes and frozen primordials	17,550
	safe-eval	S	3 / 7	◐	◐	○	○	`vm` module and mutating the context	37,090
	notevil	C+S	0 / 3	◐	◐	◐	○	meta-circular interpreter	4,387
	SandTrap [14]	S	0 / 0	●	●	○	○	`vm` module and membranes	n/a
	MIR [66]	C+S	0 / 0	●	●	○	○	shadow builtins with wrappers	6
	near-membrane	C+S	0 / 0	●	●	○	○	`vm` module and membranes	42
	AdSafe	C	0 / 3	◐	◐	○	○	static checks and wrappers	n/a
	Caja	C	0 / 2	●	●	○	○	code rewrite and frozen primordials	n/a

Table 1: Overview of our study of JavaScript sandboxes. C stands for client-side, and S for server-side. The vulnerabilities column shows the number of unfixed issues out of the total number of known breakouts. ○ means that the sandbox does not target the given security objective (SO_i), ● that the sandbox addresses the objective in a way that prevents all our attempts to jeopardize it, and ◐ that the sandbox aims to address the objective but that we could find ways of bypassing the implemented security control. The last column depicts the number of weekly downloads on npm as of 9th of October 2022.

Since the `vm` module[2] plays such a central role in many of the considered sandboxes, it is worth discussing its security assumptions in more detail. According to its documentation, it allows guest code to run with a "*different global object than the invoking code*", but at the same time, the documentation mentions: "*the* `vm` *module is not a security mechanism. Do not use it to run untrusted code*". Let us consider the following example to illustrate why running untrusted code directly in the `vm` module is a bad idea:

```
const vm = require("vm");
vm.runInNewContext(`
    obj.__proto__.foo = 23
    this.__proto__.bar = 23
`, {obj: {}});
```

In this example, we create a context containing a single empty object called `obj`. Unfortunately, the object is shared inside the new context by direct reference, allowing modifications of the host code's context by traversing the prototype chain, as seen in line 3. Moreover, this attack also works in empty contexts, because `this` can be traversed in a similar way, as seen in line 4. Thus, the `vm` module suffers from a capability leak, allowing such references to be used for reaching any Node.js' builtin.

However, it is generally accepted knowledge[3] [14, 34] that by interposing membranes for all exchanged objects, such attacks can be prevented. In Section 4.2, we discuss why this is extremely hard to achieve in practice. However, many packages in our study adopt this strategy, including `vm2` - a package with millions of weekly downloads at the time of writing. As seen in Table 1, this sandbox also has the highest number of known security vulnerabilities, possibly due to its popularity. We collect proof-of-concept breakouts for all the past vulnerabilities of this sandbox and study them in detail, together with the source code of the sandbox.

Most of the 15 breakouts are due to bugs in the implementation of the membrane, and all of them are caused by *foreign references*, i.e., references that can be used to reach builtins of the host code. Let us consider the example below:

```
const {NodeVM} = require('vm2');
let nvm = new NodeVM()
nvm.run(`
try {
  this.process.removeListener();
}
catch (e) {
  e.__proto__.__proto__.__proto__.__proto__.x=23;
}
`);
// x is 23 in the host object's context
```

The `vm2` module is selectively endowing some part of the `process` builtin object to allow the usage of listeners inside the sandbox. However, if an error is triggered in `removeListener`, the exception that is generated is a foreign reference, allowing modifications of the host object's builtins. During the study, we identified a different method in the `process` object that is vulnerable to the same attack. We

[2] https://nodejs.org/api/vm.html
[3] https://github.com/nodejs/node/issues/40718#issuecomment-960383644

https://t.me/learningnets

reported the problem, and the package's maintainers promptly deployed a fix for it. Additionally, we found a capability leak in `realms-shim` and a prototype pollution in `notevil`.

We observe that some of the vulnerabilities and exploits are specific to a given Node.js version. While the proof of concept above works in Node.js version 14.15, it does not work with newer versions, e.g., Node.js 16.12. Thus, one should consider various versions when testing language-based sandboxes.

Considering the findings in this section, we conclude that language-based sandboxes are less robust than their runtime-based counterparts. Thus, there is potential for automated approaches for finding sandbox breakout vulnerabilities.

3 Sandbox testing with SANDDRILLER

We observe that some of the patterns used in previous exploits appear benign, as seen in the code examples presented in the previous section. Thus, we hypothesize that one can use an existing code corpus to perform security testing for language-based sandboxes. Naturally, this corpus should cover as much of the language as possible, under the assumption that some bugs occur in obscure, less-tested parts of the language. At run time, we propose inspecting every reference r that could come from outside the sandbox and checking all the references in its transitive closure for signs that they may (i) be a foreign reference, (ii) point to an unprivileged operation, (iii) contain the value of a global flag we set. If any of the conditions are met, we proceed to confirm the hypothesis by attempting a sandbox breakout. If the attack is successful, the exploit is further minimized and manually analyzed.

We show our sandbox testing pipeline in Figure 2. Each code fragment in the corpus is first instrumented to include checks on relevant references, together with exploitation attempts using these references. The instrumented code is then executed on the target sandbox while a set of oracles observe security-relevant outcomes. Finally, in case of success, an analyst minimizes the exploit and identifies the root cause of the problem. Optionally, SANDDRILLER produces variations of the initial code sample by integrating ingredients from known breakouts. Recombining parts of prior exploits with benign code yields an effective way of fuzzing language engines [31], so we assume it might be useful in our case as well. Below, we discuss in detail each building block of our approach.

3.1 Instrumentor

We remind the reader that our goal is to intercept all references inside the sandbox that may point to privileged objects or methods outside the sandbox. To do so, we need to consider all the code constructs that have the potential to introduce new references in the sandbox.

When the sandbox is initialized, the only points of interaction between the untrusted and the host code are the already present references inside the sandbox: (a) endowments, i.e., white-listed APIs granted by users, and (b) global builtins, e.g., `JSON.parse`. These references can be used to invoke methods outside of the sandbox. Drawing from our experience with existing vulnerabilities, these external method invocations can introduce problematic references through (i) return values, (ii) thrown exceptions, and (iii) arguments of callbacks. Thus, we propose instrumenting the following code constructs:

```
1  // replace every function call foo() with
2  (let temp = foo() & checkReference(temp) & foo);
3
4  // replace every try-catch with
5  try {
6  } catch(e) {
7      checkReference(e);
8  }
9
10 // replace every function definition with
11 function foo() {
12     checkReference(arguments);
13     checkReference(this);
14 }
```

At each instrumentation site, SANDDRILLER inserts checks on the newly introduced reference in that scope, e.g., a function's argument. While we do not provide any guarantees that these transformations are semantics-preserving, we note that this is not in line with our goal anyway, e.g., code with slightly different semantics can still be useful for finding bugs, as we discuss in Section 3.3. Moreover, tools like Jalangi [56] use similar transformations for implementing dynamic analysis.

For each reference check, SANDDRILLER first computes the transitive closure of that particular reference, i.e., it traverses all the reachable references by iterating through its properties recursively until a fixed point is reached.

For each entry in the transitive closure of the checked reference, SANDDRILLER performs three types of runtime checks:

- **Read secret flag:** checks whether the entry contains the value of a high-entropy variable defined in the global scope of the host code. This check verifies if the global scope of the host process is included in the transitive closure of the reference, hence the high-entropy value can be accessed directly.
- **Access the require method:** checks whether the entry's constructor is used to get a pointer to the `require` method. This check is inspired by some of the exploits we observe in Section 2: when invoking the constructor of a foreign reference, sometimes `this` binds to a global object that contains a reference to the `require` method.
- **Different root prototype:** checks whether the reference has a different root prototype than the root prototype of the sandbox. This final check is a heuristic we use to identify foreign references. While this check has the potential to be noisy, in the next section, we discuss how we validate that a reference flagged by this check can indeed be used to escape the sandbox.

https://t.me/learningnets

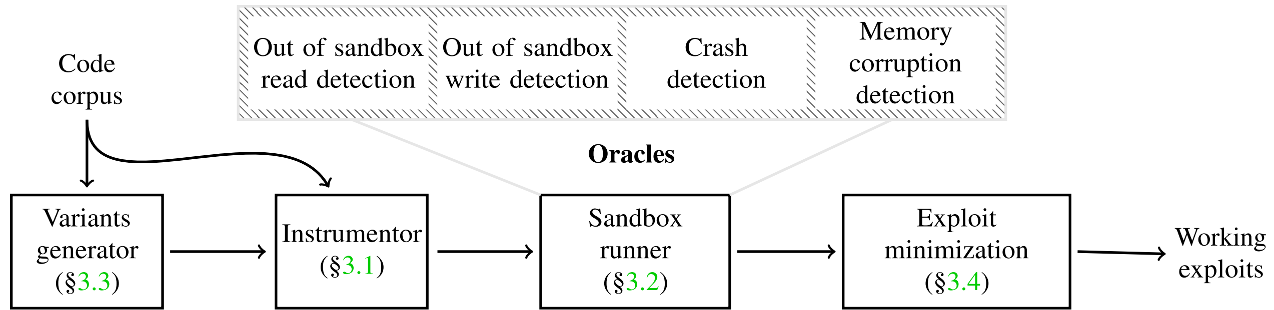

Figure 2: SANDDRILLER, a fully automated approach for testing language-based JavaScript sandboxes. The arrows show how code samples from the corpus are processed through the testing pipeline.

3.2 Sandbox runner

The sandbox runner uses a pool of processes, each process running a separate test. It also allocates a time budget for each test, after which the corresponding process is killed. In such cases, or when the test process crashes on its own, a new process is spawned to keep the pool's size constant.

To be able to run a piece of code from the corpus inside the sandbox, we need to make sure it has the right endowment, e.g., the builtin `console` object. We note that ensuring all programs in the corpus correctly run inside the sandboxes is an extremely hard problem, which is out of the scope for this work. Moreover, by providing an extensive list of endowments, in case a sandbox under test suffers from a capability leak, as discussed in Section 2, SANDDRILLER will produce a violation for each usage of the endowed API. Hence, we decide to be conservative and only endow two APIs: a surrogate for the `require` method that is responsible for bringing additional helper code inside the sandbox, a method called `leak` for sending results back outside the sandbox, e.g., notify the host code that the private flag was successfully read.

At the start of each test, a global flag is set in the global scope of the host code. The test code is then executed inside the sandbox while monitoring invocations of the `leak` method for any unprivileged read that occurred in the sandbox, in which case a security error is logged. If the "different root prototype" check for a reference inside the sandbox succeeds, the instrumented code attempts to use that reference to set a flag on the root prototype of the host code. At the end of the test's execution, the runner verifies if this flag is present on the root prototype and logs a security error if that is the case.

For client-side sandboxes, the process runner runs each instrumented test in a fresh browser window. The `leak` method transmits the outcome of the test outside of the browser to the process runner, which further logs it. Nonetheless, the check for polluted root prototypes is performed inside the browser since we do not expect our tests to be able to break out from inside the browser into the Node.js space. Also, the performance measurements are performed inside the browser to measure only the execution of the test, not also the time spent for initializing the browser.

For detecting possible memory violations, we first use a coarse-grained oracle that logs any possible crash. In addition, we also use a finer-grained oracle: we compile Node.js with memory error monitoring and log any invalid memory accesses. After any such violation, SANDDRILLER replenishes the process pool to keep the number of workers constant.

Upon completion, SANDDRILLER logs seven possible outcomes for a test: (1) run without errors, (2) run with runtime error, (3) instrumentation error, (4) security error, (5) timeout, (6) hard crash of the process, (7) memory violation. It also logs the number of oracle checks performed during execution and the time it takes to execute the test.

3.3 Variant generator

We noticed that nine out of 27 known exploits in Section 2 use the following pattern:

```
1 try{
2     ...
3     throw x=>x.constructor("return process")();
4 }catch(e){
5     process = e(()=>{});
6 }
```

From a nested calling context, a custom closure function is thrown for encapsulating the scope in which the exception was thrown. In the catch clause, the closure is invoked to obtain a reference to a global object of the host code. Considering the prevalence of this pattern, we hypothesize that it captures an important challenge of building language-based sandboxes: handling complex nested scopes.

Thus, we propose creating variants of the tests in the corpus by wrapping their code in a `try-catch` block and throwing an exception on the first line of every function definition, i.e., a coarse-grained approximation for a new scope. For example, for a test file with four function definitions, SANDDRILLER produces four variants. Each variant consists of the original code wrapped in a `try-catch` block and a `throw` statement in one of the functions. The `catch` clause invokes the thrown closure, like in the listing above.

https://t.me/learningnets

We note that, as opposed to other fuzzing techniques, the proposed way of augmenting the original code corpus is entirely deterministic. For each test, we create a fixed number of variants that is the same across multiple executions. One may argue that the presented variant generator overfits to the studied vulnerabilities, and this may indeed be the case. However, we aim to explore the feasibility of recombining known exploit fragments with benign code corpora as a way of boosting the effectiveness of testing language-based sandboxes.

3.4 Checking outcome of tests

In the last step of the pipeline, we manually study interesting tests that exhibit either security violations, memory corruptions, or hard crashes. The way SANDDRILLER constructs instrumented test cases guarantees that every flagged test triggers a security-relevant operation, e.g., prototype pollution. That is, our testing methodology has no false positives because each instrumented test is a proof of vulnerability. SANDDRILLER only flags tests for which it observes a read or write outside of the sandbox, a crash of the runtime, or a memory violation. Nonetheless, it can happen that the security-relevant outcome is not directly caused by the instrumented test alone, but by complex interactions with SANDDRILLER's code base or with the code fragments introduced by the variant generator. Thus, the first step in analyzing an offending test case is to run it in a minimal setup, i.e., a sandbox with no endowments.

Nonetheless, the instrumented test case in this step is rather complex, so it is not feasible to use it for reporting security bugs or for understanding the root cause of the problem. Hence, we propose using delta debugging [72] for minimizing the instrumented sample. While preserving the security-relevant outcome as invariant, e.g., the code should still pollute the global scope after minimization, we reduce the size of the code as much as possible. Once minimized, we store each sample for further grouping with similar tests or for reporting it to the maintainers of the sandboxes. We also study whether the minimized exploit is portable across all the considered Node.js versions. For grouping the exploits, we use our best judgement, mostly by analyzing the employed language features. For example, we group two tests using infinite recursion wrapped in a `try-catch` block as likely being produced by the same bug, even if the concrete syntax used for producing the recursion is different in the two tests.

3.5 Implementation

We implement SANDDRILLER in about 2,000 lines of JavaScript code, using `esprima` parser [4] for instrumentation and for generating variants. We use `deltajs` [5] for exploit minimization, but often observe cases in which it gets stuck, and need to complement it with manual delta debugging. For testing client-side sandboxes, we employ Puppeteer [6], an orchestration framework for headless Chrome browser, to spawn a browser instance for each test. We use Address Sanitizer [7] as our memory violation oracle.

4 Evaluation

To illustrate the effectiveness of SANDDRILLER, we present an empirical study involving multiple sandboxes, showing that our approach is able to find vulnerabilities in most of them. We start by discussing our empirical setup, then we present the quantitative results of our study in Section 4.1 and showcase the most important vulnerabilities found by SANDDRILLERin Section 4.2. Finally, we discuss our findings in Section 4.3.

Setup. In our study, we consider six language-based sandboxes from Section 2 that we could easily set up with our testing infrastructure: `vm2`, `realms-shim`, `safe-eval`, `near-membrane`, `AdSafe`, and `ses`. Thus, we exclude sandboxes that require us to specify security policies, e.g., `SandTrap` or `MIR`. Due to `Caja` being discontinued since early 2022 and the lack of documentation about its isolation component, we conclude that `Caja` is incompatible with our setup.

We run SANDDRILLER against the latest version of each sandbox, on three different Node.js versions: 14.15, 15.12, and 16.12. Node.js uses parallel releases, so at any given moment, there are multiple recent releases, all having a different feature set. At the time of developing our prototype, 15.12 and 16.12 were two "current" releases from 2021, and 14.15 was a "long term support (LTS)" release from the same year[8]. As code corpus we use ECMAScript conformance tests (ECMA) and V8 unit tests (V8), containing 41,034 and 5,572 tests, respectively. We create at most five variants of each program in the corpus using the approach described in Section 3.3. Finally, we use a process pool of size 16, and a timeout of 10 seconds for each test. We use our infrastructure to run SANDDRILLER against the above-mentioned sandboxes. We run our experiments on a server with 64 Intel Xeon E5-4650L@2.60GHz CPU cores, 768GB of memory, running on Debian GNU/Linux 10.

As a sanity check for our approach, we run SANDDRILLER on all the exploits for known vulnerabilities collected in Section 2 and show that it can detect foreign references in all of them. However, we note that this is a poor proxy for showing the success of our technique, as these exploits were written by humans. While certain parts of these exploits may be available in the considered corpus, most of them are overfitted to the identified bug, so SANDDRILLER could not synthesize them using the benign samples. Nonetheless, in Section 4.2 we show previously-unknown vulnerabilities for which SANDDRILLER can automatically synthesize an exploit.

[4] https://esprima.org/
[5] https://github.com/wala/jsdelta
[6] https://github.com/puppeteer/puppeteer
[7] https://github.com/google/sanitizers/wiki/AddressSanitizer
[8] https://nodejs.org/en/blog/year-2021/

https://t.me/learningnets

4.1 Quantitative analysis

In this section, we discuss the high-level results of our testing, deferring the presentation of the security findings to the following section.

Table 2 shows the test outcomes across all the experiments we performed, on all the selected Node.js versions, including both the test cases from the original corpus and their variants. SandDriller finds at least one security error in four out of five considered sandboxes, and hard crashes in three of them. `ses` is the only sandbox for which our tool could not find a breakout. On the contrary, `ses` has the highest number of runtime errors and timeouts of all the sandboxes in our evaluation. This suggests that freezing the intrinsics, a technique only used by `ses` in our evaluation, might provide better isolation at the price of preventing some legitimate user code from executing. `ADSafe`, `near-membrane` and `safe-eval` have by far the highest number of security errors in our study. After manual inspection, we conclude that this is because these three sandboxes fail to prevent some trivial ways of escaping the sandbox, e.g., using `this` as a foreign reference. We believe that for `ADSafe` this is because this sandbox relies on JSLint[9] to enforce certain static transformations, e.g., removing direct accesses to the prototype objects. Recent versions of JSLint do not perform such transformations, hence, breaking the invariant of the sandbox. The number of hard crashes is small overall, but this is to be expected in a language like JavaScript that adopts a no-crash policy. Similarly, the memory corruption oracle produces few violations for a handful of tests in our corpora, e.g., allocation-limit.js from ECMA. These tests attempt to allocate a very large `ArrayBuffer`, which causes Address Sanitizer to stop the execution of the program with the message "requested allocation size exceeds maximum supported size". This shows that such oracles are too coarse-grained for our testing problem, as they fail to capture security bugs that occur in upper layers.

Considering that the tests in our corpus run without a problem outside of the sandbox, but many of them fail to run in our setup, we inspect some of the failures to see why that is the case. Most of the cases we analyzed are due to missing objects in the global scope, but we also found cases in which the sandbox did not support certain language features, e.g., mutating the root prototype when running inside `ses`. Thus, we conclude that the number of errors can be further reduced by extending the list of endowments, but that SandDriller finds a significant number of security errors, despite the high number of tests that fail to run with our testing pipeline.

In Table 3, we present the security-relevant outcomes grouped by Node.js version, data set, and the use of our variant generator. We do not include `AdSafe` in this table because it is a client-side sandbox that we run inside a headless browser, hence we do not observe any variations across different Node.js versions. We find that the number of security violations and hard crashes increases when using the variant generator. However, we warn the reader that this increase is partially an artifact of our variant generator, i.e., if the original test triggers a security error, most of its variants probably do so too. Nonetheless, this should at most magnify the number of violations with a factor of 5 (the number of variants), but we see cases like `vm2` on Node.js 16.12 where this increase is larger. We also note that both data sets in our corpus can find security bugs in the affected sandboxes. Hence, while the diversity of the corpus' code plays a critical role in the success of SandDriller, we conclude that our approach is generic enough to work with various sources of input code samples. Finally, we observe that there are important differences across Node.js versions, validating our observation from Section 2 that some exploits only work on a specific version of the runtime.

Oracle checks. Figure 3 shows the number of oracle checks against the selected sandboxes, using both ECMA and V8 data sets in our corpus, on a specific Node.js version 16.12. In all sandboxes used in our evaluation, we find that the peak of all distributions on both data sets is located close to zero oracle checks. This is probably due to the large number of runtime errors we report in the previous section. Moreover, there is a significant number of tests with approximately 20,000 oracle checks. We hypothesize that this is due to test cases in which SandDriller recursively checks the transitive closure of the global object. This object contains references to all the intrinsics, and thus, recursively traversing all their properties triggers a lot of oracle checks. Nonetheless, the list of intrinsics enabled by default inside the sandbox is much larger in `vm2`, `near-membrane`, and `safe-eval` than in `ses` and `realms-shim`. For example, `SharedArrayBuffer` is available by default in the first three sandboxes but not in the last two. Finally, even though the number of oracle checks appears low in `realms-shim`, in the following section, we show that SandDriller finds important security breakouts in this sandbox, so the number of oracle checks is a poor proxy for effectiveness.

Performance. To understand the total time it takes for SandDriller to run the tests in the corpus, Figure 4 shows the distribution of testing time for a specific Node.js version 16.12. These results show that most of the test cases finish in less than 200 ms, independent of the sandbox they are run on. This is not entirely unexpected, as the two data sets we consider contain mostly simple unit test programs that are expected to run very fast. It is interesting to observe that the three sandboxes with a large number of oracle checks in Figure 3 are also the slowest ones. This suggests that oracle checking is a non-negligible slowdown factor introduced during instrumentation. Nonetheless, one can run thousands of test cases with SandDriller in a few minutes, showing its potential to be used in production.

[9] https://www.jslint.com/

https://t.me/learningnets

Sandbox	Data set	Without errors	Runtime error	Timeout	Security error	Hard crash	Memory corruption
vm2	ECMA	111742	117315	35311	26	18	6
	V8	15515	14904	6323	56	4	0
realms-shim	ECMA	115750	119408	45008	60	0	6
	V8	15085	15030	9710	55	0	0
safe-eval	ECMA	89521	115106	38426	27482	18	6
	V8	8043	14749	7398	8054	4	0
ses	ECMA	113834	119364	47759	0	0	6
	V8	15173	14986	9740	0	0	0
near-membrane	ECMA	85410	112964	37414	29926	0	6
	V8	8053	14706	7064	9644	4	0
ADSafe	ECMA	1999	20219	4419	44441	0	0
	V8	516	7536	6589	846	0	0

Table 2: The outcome of running the programs in the corpus inside the considered sandboxes. We note that the numbers also include variants of the programs in the corpus generated using the approach described in Section 3.3. The last three columns of the table depict the security-relevant results detected by SANDDRILLER. We highlight sandboxes with known vulnerabilities or for which maintainers responded that the project is a prototype.

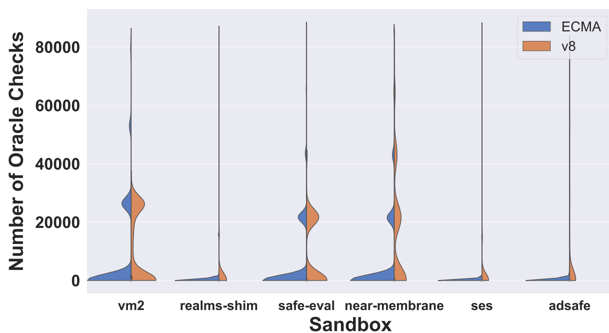

Figure 3: Distribution of the number of oracle checks per test in different sandboxes. We show results for the two data sets considered in our corpus: ECMA and V8.

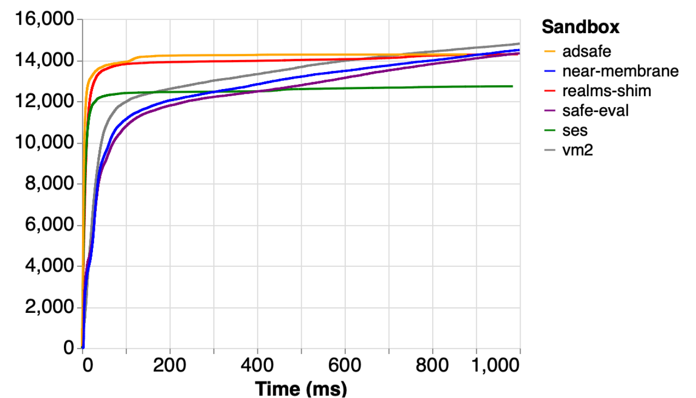

Figure 4: Cumulative distribution function for the execution time of tests. The data corresponds to Node.js version 16.12.

4.2 Qualitative analysis

As mentioned earlier, manually grouping the identified sandbox breakouts based on the root cause is a tedious, error-prone process. Considering the large effort required to maintain an open-source project, we decided to be selective when reporting vulnerabilities and avoid overloading maintainers with all the reports at once. We describe below our experience with vulnerability disclosure and present the most important sandbox breakouts. We warn the reader that a single advisory for a given sandbox may correspond to multiple reported attack vectors. In Table 4, we depict the confirmed vulnerabilities we reported, and their time of reporting and fixing, if applicable. In the last column, we list the concrete payload we reported. As observed, some security advisories correspond to multiple payloads since the sandbox maintainers handled them as a batch. We now proceed by describing a bug we found in Node.js' `vm` module, which affects most of the considered sandboxes.

Node.js. During our manual inspection, we find a recurring, hard-to-explain pattern in many of our exploits. We show a distilled version of this pattern in Figure 5. The code triggers an infinite recursion exception, which acts as a foreign reference for sandbox breakout. The intriguing part of this example is line 8, which is an essential ingredient for the exploit to work. This unused line creates an exception and accesses its stack property. After carefully debugging this code, we report that if this line is not present, the try-catch block executes only once, but if the line is present, the block executes 498 times in our setup. We traced this bug to a commit from 2019[10],

[10] https://github.com/nodejs/node/commit/6f7005465a

https://t.me/learningnets

			Original corpus		Variants	
Node.js version	Sandbox	Data set	Security error	Hard crash	Security error	Hard crash
14.15	vm2	ECMA	1	2	6	4
		V8	7	1	11	1
	realms-shim	ECMA	4	0	8	0
		V8	12	0	4	0
	safe-eval	ECMA	3154	2	6214	4
		V8	812	1	2079	1
	ses	ECMA	0	0	0	0
		V8	0	0	0	0
	near-membrane	ECMA	2873	0	6877	0
		V8	793	1	2476	1
15.12	vm2	ECMA	1	2	4	4
		V8	7	1	12	1
	realms-shim	ECMA	4	0	12	0
		V8	12	0	4	0
	safe-eval	ECMA	3142	2	5484	4
		V8	738	1	2071	1
	ses	ECMA	0	0	0	0
		V8	0	0	0	0
	near-membrane	ECMA	3075	0	6360	0
		V8	793	1	2476	1
16.12	vm2	ECMA	1	2	13	4
		V8	8	0	12	0
	realms-shim	ECMA	17	0	15	0
		V8	18	0	5	0
	safe-eval	ECMA	3957	2	5531	4
		V8	475	0	1879	0
	ses	ECMA	0	0	0	0
		V8	0	0	0	0
	near-membrane	ECMA	3581	0	7160	0
		V8	626	0	2374	0

Table 3: Violations reported by SANDDRILLER in the three considered Node.js versions, using the corpus and its variants. We highlight sandboxes with known vulnerabilities or for which maintainers responded that the project is a prototype.

in which error stack traces were refactored. Our hypothesis is that when preparing the stack trace, the C++ code invokes a `ReThrow()` function, which spawns additional unwrapped exceptions inside the `vm` module, acting as foreign references. This potential bug breaks the assumption that lies at the core of most sandboxes we consider: augmenting the `vm` module with membranes yields a secure sandbox. That is because membranes are deployed at the interface between guest and host code, but in the example above, the exception is spawned as a result of invoking custom functions defined inside the sandbox. Hence, no proxy is deployed in line 6 to mediate accesses on the problematic exception. We warn the reader about the semantic gap between the source of the bug (low-level C++ code inside Node.js internals) and the entity under test (sandbox's code implemented in JavaScript). This gap makes writing and maintaining language-based sandboxes an extremely difficult task, requiring developers to account for the unpredictable behavior in the lower level.

We reported this bug to the Node.js team and described it as a security problem with significant impact in the ecosystem. We were very surprised by the response we got from them: "*This looks like a vulnerability on those sandboxes and not the* `vm` *module itself as it is clearly marked as not being security sandbox. We clearly do not provide any security guarantee on the behavior of* `vm`.". This means that the affected sandboxes must patch their implementations in a very cumbersome and error-prone way: preventing access to all methods that prepare a stack trace for an exception. Deploying such a fix has serious side-effects on the usability of these sandboxes as well: benign code that is willing to use this API is suddenly prevented

https://t.me/learningnets

Sandbox	Vulnerability	Date of reporting	Date of fixing	Details about the payloads
isolated-vm	CVE-2021-21413	8th of February 2021	12th of February 2021	• capability leak
vm2	CVE-2021-23449	15th of September 2021	12th of October 2021	• import keyword • custom stack traces
vm2	CVE-2021-23555	25th of November 2021	8th of February 2022	• vm's stack property issue
vm2	Issue #285	29th of April 2020	29th of April 2020	• custom toString() method on listener objects
realms-shim	CVE-2021-23594	10th of December 2021	n/a	• custom stack trace
realms-shim	CVE-2021-23543	10th of December 2021	n/a	• vm's stack property issue
SandTrap	GHSA-xx7r-mw56-3q2h	22nd of September 2021	11th of November 2021	• import keyword • vm's stack property issue • custom stack traces
jailed	CVE-2022-23923	4th of January 2022	n/a	• direct access to powerful builtins
notevil	CVE-2021-23771	4th of January 2022	n/a	• bypass restriction on property names

Table 4: Confirmed vulnerabilities in response to our findings. Each item point in the last column represents a sandbox breakout payload that we reported to the maintainers of the sandboxes.

```
1  const vm = require("vm");
2  vm.runInNewContext(`
3  function test() {
4   try {
5     test();
6   } catch (e) {
7     e.__proto__.__proto__.__proto__.polluted = 23;
8     new Error().stack;
9   }
10 }
11 test();`);
```

Figure 5: A non-trivial escape from the vm module, caused by a bug in Node.js. If we remove the highlighted line, which appears superfluous at first, the escape stops working. The bug is present in all three Node.js versions.

```
1  const {VM} = require("vm2");
2  let vmInstance = new VM();
3  vmInstance.run(`
4  try {
5     Object.defineProperty(
6        RegExp.prototype,
7        Symbol.match, {
8           get: function () {
9              new Error().stack;
10             "x".match(/a/)
11          }
12       })
13    "x".match(/a/);
14 } catch (e) {
15     e.__proto__.__proto__.__proto__.polluted = 23;
16 }`);
```

Figure 6: A sandbox breakout vulnerability in vm2 found by SandDriller. It is a variation of the exploit in Figure 5 that only runs in Node.js 16.12 and 15.12, but not in 14.15.

from executing inside the sandbox. We believe that this is a very unfortunate development and that the community should be aware of this state of affairs. Thus, we worked with the Node.js team to improve the documentation and send this message more explicitly[11].

Before describing additional findings in the individual sandboxes, it is worth reflecting on this important empirical result. We conclude that language-based sandboxes are only as effective as their building blocks, i.e., if one of these blocks misbehaves and there is no obvious way of fixing it, one must either replace that block or expand the existing assumptions to accommodate the misbehavior. In our case, this means augmenting the membrane-based solution with additional restrictions on the code running inside the sandbox.

Vm2. We reported a total of four sandbox breakout vulnerabilities for this sandbox, one already discussed in Figure 1. Initially, we faced very slow response times from the maintainers, and we were even offered to take over the maintainer role for this package. Considering that this is a package with almost 500 dependents and three million weekly downloads, we

[11] https://github.com/nodejs/node/commit/86fba23b1f7524889205a1c02e8f9010ad61998e

https://t.me/learningnets

```
const {VM} = require("vm2");
let vmInstance = new VM();
vmInstance.run(`
 Object.prototype.get = 0;
 Object.getOwnPropertyDescriptor(this, "VMError")
      ['get'];
`);
```

Figure 7: A hard crash in vm2 found by SANDDRILLER. It works in all the considered Node.js versions.

```
import Realm from 'realms-shim'

let realm = Realm.makeRootRealm();
try {
  realm.evaluate(`
    Error.prepareStackTrace = function (e, st) {
      st.__proto__.__proto__.polluted = 'success'
    };
    x;`);
} catch(e) {
    // we do not even need to print e
}
```

Figure 8: CVE-2021-23543, a sandbox breakout vulnerability in realms-shim found by SANDDRILLER. The exploit uses a custom stack trace method to escape the sandbox.

found this attitude distressing, especially when considering the recent supply chain attacks. After contacting a company specialized in disclosing security vulnerabilities, a fix was deployed, and a CVE was issued to warn the users about the potential security risks. At the time of writing, all breakouts are already fixed. We draw the reader's attention to the bug in Figure 6, for which the authors deployed a cumbersome fix consisting of heavy refactoring of the sandbox's code[12]. While this bug is caused by the vm module's erratic behavior, there are a couple of important differences compared to the exploit discussed before. First, this latter one only works on two out of three considered Node.js versions, showing the importance of testing against different versions of the runtime. We notice that the spurious line with the stack access is in a different position, and if one attempts to move it after the recursive call in line 10 or into the catch block in lines 14-16, the exploit stops working. The deployed fix for the bug is to tightly control access to stack traces from inside the sandbox and to add a similarly spurious call to the stack property inside the sandbox's code[13]. In Figure 7, we show a hard crash of Node.js caused by untrusted code running inside the vm2 sandbox. The exploit first defines an invalid getter function on the root prototype, and then it retrieves a property descriptor of the VMError object, a custom intrinsic defined by the sandbox.

[12] https://github.com/patriksimek/vm2/commit/2353ce60351c50379b8d1daab05812c4db634162

[13] https://github.com/patriksimek/vm2/blob/392f126b18d5f6e1ea9300a2176707fc852da863/lib/setup-sandbox.js#L183

When the get property of the descriptor is accessed, Node.js unexpectedly crashes with "Invalid property descriptor". It is worth mentioning that the same code, when run outside the sandbox or inside another sandbox, does not crash the runtime. Hard crashes are rare in JavaScript, and they can be used to perform powerful denial-of-service attacks. However, our initial motivation was to look for crashes as a symptom of memory corruption. After manually analyzing the reported crashes, we conclude that all of them are caused by assertions in the C++ code and not by illegal memory accesses.

Realms-shim. We initially exchanged several emails with the maintainers of realms-shim to better understand their threat model. A significant part of the discussion of security objectives in Section 2 developed during these interactions. The maintainers of this sandbox told us that they shifted their interest towards ses, but realms-shim was still quite active at the time. We thus considered both ses and realms-shim in our study. After reporting several vulnerabilities we identified, they decided to stop maintaining that package and mark it as deprecated. Moreover, Snyk issued two CVEs for our findings in this package to send an even stronger message about the dangers of using it. In Figure 8, we show one of the two breakouts SANDDRILLER found. It declares a custom stack trace method and then triggers a runtime error by referring to a non-existent variable x. Though the stack trace of the error is never explicitly printed, the payload for polluting the global object in line 7 is triggered by the sandbox's internals.

Near-membrane. Even though we find multiple vulnerabilities for the server-side version of this sandbox, we only reported one to first understand developers' assumptions. The maintainers replied that they are aware of the problem, but "*this is not a library that is used internally nor do we have any customers externally using this library*". We believe that the documentation of the package should make these assumptions more clear, e.g., communicating that the project is a prototype and nobody should use it in practice.

SandTrap. While we did not test this sandbox directly in our study, we checked if any of the exploits we found for the other sandboxes work against SandTrap. Initially, we had a hard time setting up the tool, but after getting in touch with the authors, they assisted us by creating a sample policy and clarifying the threat model. Three of the exploits identified for other sandboxes worked against SandTrap as well, showing that developers of independent projects tend to make the same mistakes or overlook the same important assumptions about the underlying infrastructure. The authors promptly fixed two of the breakouts and issued a security advisory (GHSA-xx7r-mw56-3q2h). For the last one, they are still searching ways of mitigating the unpredictable behavior of the vm module.

https://t.me/learningnets

4.3 Discussion

Our results show that SandDriller can detect non-trivial bugs in multiple sandboxes, with modest testing effort. Most critical vulnerabilities we discovered are sandbox breakouts that violate the objectives (SO_1, SO_2). We find that language-based sandboxes often fail to deliver on their promise because of intricate corner cases involving obscure parts of the language or poorly documented behavior of the runtime. Considering the identified issue in the C++ implementation of Node.js' `vm` module, we speculate that SandDriller may also be applicable to runtime-based sandboxes that allow foreign references, which are an essential part of our methodology. We believe that sandbox developers should consider integrating SandDriller in their development process to detect vulnerabilities automatically early on in the release life cycle. The practitioners we interacted with were often intrigued that we found the bugs automatically, wanting to hear more about our tool. This shows that there is a real need for an approach like ours.

While SandDriller has no false positives by construction, i.e., every alert corresponds to code that breaks a security invariant, it cannot guarantee the absence of false negatives, a limitation inherent to testing. That is, SandDriller can only show the presence of sandbox breakouts, not their absence. Thus, our approach is not a way to certify a given release, but a sanity check that the sandbox can deal with real-world code, employing a variety of language constructs. We note that this is in stark contrast with the objectives of prior work that aims to verify the code of sandboxes [43,44,54,62].

Our work targets supply chain components in isolation, not entire systems. Nonetheless, the identified vulnerabilities allow complete breakout of popular sandboxes, which are often used in real-world systems. Since we are not aware of an ethical way of verifying the vulnerability of live systems to our payloads, we recommend users of these sandboxes to migrate to the latest version as soon as possible. Moreover, considering the inability of language-based sandboxes to defend against denial-of-service attack vectors (SO_3, SO_4), we recommend sandbox users to follow the defense-in-depth paradigm and to deploy additional security controls. For example, the maintainers of `ses` told us that they recommend deploying an additional layer of security similar to the one advocated by TreeHouse [32] and BreakApp [65] where the untrusted code is run in a separate thread or process to prevent resource exhaustion attacks. Thus, by combining language- with runtime-based isolation, one can achieve both satisfactory security guarantees and fine-grained access to resources.

Moreover, we find important similarities between testing sandboxes and testing language engines. Future work should explore more sophisticated ideas from this adjacent research area. For example, one could define more complex mutations that reuse a larger part of known exploits [31] and add a feedback loop carrying runtime information about the sandbox under test to inform the mutation selection. The used corpora can also be partitioned across language features to ensure uniform coverage of the grammar [15] or to prioritize new language features through space reduction. Nonetheless, it is essential that such ideas are combined with our novel domain-specific oracles to produce actionable sandbox breakouts.

Our approach is extensible in multiple ways. First, to support new language features, the input corpus should simply be augmented with code containing these features. Second, to consider new oracles, SandDriller only needs a direct communication channel with the oracle to collect possible violations. Third, one can also extend the list of security objectives and perform an additional study of sandboxes with respect to the new objective. However, we expect objectives to rarely change, only after paradigm changes like the recent one involving microarchitectural side-channels [35]. Thus, to include this new assumption, one should first study the resilience of the sandboxes to a simple speculative execution attack (SO_5) and add a new column in Table 1. Then, using a feedback channel from CPU-level, one should define an additional oracle to detect out-of-order executions. SandDriller can then test sandboxes that claim to prevent speculative execution attacks to find new attack vectors against them. The discussed scenario is hypothetical, since none of the considered sandboxes prevent such side channels.

5 Related Work

In this section, we survey closely-related work on building and verifying JavaScript isolation techniques, fuzzing JavaScript engines, and server-side JavaScript security.

Verifying JavaScript Isolation. Politz et al. [54] propose a type-based static analysis for identifying bugs in JavaScript sandboxes. They show that it can detect multiple types of bugs in ADSafe, one of the first language-based sandboxes for JavaScript. Taly et al. [62] propose a slightly different approach by providing a rigorous definition for the reference-monitoring functionality of a sandbox. By analyzing the operations performed by the sandbox using JavaScript's semantics and the presented definition, the authors could identify multiple bugs in the analyzed sandboxes. Similarly, Maffeis et al. [45] show how verifying isolation on a subset of the language can be used to detect bugs in real-world sandboxes. Also, Maffeis et al. [44] introduce authority safety, an objective that language-based sandboxes should achieve to provide sound isolation. Finifter et al. [25] show how state-of-the-art sandboxes fail to prevent access to methods defined on the root prototype. We are the first to present a dynamic analysis-based approach for testing language-based sandboxes. There are also approaches for verifying the confinement of JavaScript code after it is compiled by the just-in-time (JIT) compiler. Maisuradze et al. [46] propose a fuzzing

https://t.me/learningnets

approach for finding non-blinded user constants in the code produced by the JIT. Park et al. [51] present sophisticated attacks that are enabled by unsafe handling of user's code inside the JIT. While such bugs can compromise the security of language-based sandboxes, we consider them out of scope.

JavaScript Isolation. There are many proposed techniques for isolating JavaScript code deployed in various levels of the software stack. Language-based sandboxes often rely on two important building blocks: proxies [19] and membranes [47], which are popular ways of implementing reference monitoring in JavaScript. JSand [13] uses membranes and server-side maintained policies to enable the creation of web mashups. NodeSentry [29] and DecentJS [33] propose using membranes and security policies for reducing the capabilities of JavaScript code. JaTE [63] heavily relies on the semantics of the `with` statement. More recently, MIR [66] advocates for interposition using reference monitors deployed in the local scope to shadow global builtins. Similarly, Ferreira et al. [24] propose a permission system enforced using program rewriting, i.e., it inserts runtime checks at program locations that use relevant property names. Ahmadpanah et al. [14] propose augmenting Node.js' `vm` module with membranes. Ko et al. [34] introduce the SecureJS compiler for enforcing memory isolation in JavaScript runtimes that do not offer this feature. There are also coarser-grained techniques for JavaScript isolation: AdSentry [22] advocates for using additional instances of the runtime, Treehouse relies on web workers [32], while BreakApp uses operating system processes [65]. JSISOLATE [73] proposes modifying the browser to intercept relevant operations performed by the untrusted code. Isolation of JavaScript code can also be done at lower levels: sandboxing the browser with Native Client [71], parts of the browser using RLBox [48] to prevent dynamic compromise [11], or even adding hardware support for isolation using EIRM [64], IMIX [26], or Enclosure [28]. Recently, Wyss et al. [68] proposed using system call filtering to reduce the capabilities of install-time hooks, which are extensively used in supply chain attacks. We note that many of the techniques above offer robust isolation of JavaScript code, but practitioners rely instead on cheaper solutions that allow sandbox breakout through foreign references. SANDDRILLER is thus a pragmatic approach for a real-world problem identified in the open-source ecosystem. Bhargavan et al. [16] study the dual problem of protecting guest code from untrusted hosts. They propose DefensiveJS, a subset of the language that guarantees the integrity of the JavaScript code. SANDDRILLER is not directly applicable to this case, but we believe that a similar solution based on foreign reference oracles can detect unexpected references that can be mutated by the host, e.g., `toString` builtin method.

Fuzzing JavaScript Engines. There is a growing interest in testing JavaScript engines using fuzzing techniques. Holler et al. [31] use the language's grammar and code fragments from previous exploits to identify unknown bugs. Park et al. [50] propose a more sophisticated way of recombining fragments from known exploits, using aspect-perceiving mutation. Aschermann et al. [15] augment grammar-based fuzzing with coverage information, while Lee et al. [37] use machine learning-based language models to generate new test cases. Patra et al. [53] infer probabilistic, generative models from input JavaScript corpora. Similarly, Ye et al. [70] train a deep learning model on the specification of the language and further use this model to generate tests. It further uses these models for generating code that is syntactically similar to the one in the corpus. Han et al. [30] enforce semantic constraints on the variants produced during fuzzing to minimize the number of invalid programs generated. Veggalam et al. [67] use evolutionary algorithms to increase the performance of grammar-based fuzzers. As discussed in Section 4.3, testing language-based sandboxes has many similarities to engine fuzzing, and thus, many of these ideas can be used to extend our work.

Security of Server-Side JavaScript. Running JavaScript outside the browser is an important recent trend, with Node.js and its ecosystem npm at the center of this movement. Zimmermann et al. [74] study the two main risks associated with npm packages: depending on vulnerable code and supply chain attacks. They show that the average npm package relies on 79 other packages and on 39 maintainers. There is an increasing body of work studying important classes of vulnerabilities in Node.js libraries: injection vulnerabilities [27, 59], ReDoS [20, 21, 42, 58], supply chain attacks [23, 24, 52, 68, 74], prototype pollution [40, 41, 57], hidden property abuse [69], low-level code [18, 61], code bloat [12, 36]. Another emergent threat is the reliance of these libraries on WebAssembly, which has the potential to introduce new vulnerabilities [38] and to reduce the performance of existing security tools [55]. Li et al. [40, 41] propose using object dependency graphs, a novel representation of JavaScript code, for statically detecting vulnerabilities. Koishybayev et al. [36] advocate for debloating npm packages using static fine-grained call graph analysis. Nielsen et al. [49] use call graphs for more effective software composition analysis. Duan et al. [23] detect malicious packages using a multi-stage hybrid program analysis. Staicu et al. [60] propose TASER, a technique for extracting security specifications for npm libraries, which are further used to increase the recall of static analysis. Davis et al. [21] augment the Node.js runtime with a defense against CPU-based denial-of-service attacks. In Chapter 2, we discuss three recently-proposed isolation techniques for server-side JavaScript and analyze their security objectives: BreakApp [65], MIR [66], and SandTrap [14]. None of the works above tests popular sandboxes available in the npm ecosystem. Recently, Bhuiyan et al. [17] proposed an extensive benchmark suite of known vulnerabilities in server-side JavaScript, including several vulnerabilities identified by us.

https://t.me/learningnets

6 Conclusion

In this paper, we present SANDDRILLER, a dynamic analysis technique for testing language-based JavaScript sandboxes. We show that it is effective at finding bugs in real-world code by uncovering 12 sandbox breakouts in open-source sandboxes, corresponding to eight high-severity advisories. Our solution is lightweight and, thus, easy to integrate into existing development processes to detect sandbox breakout vulnerabilities before each software release. Additionally, we identify an unusual dynamic between Node.js' maintainers and the sandbox creators: there is a mismatch between the latter's expectations and the former's disponibility towards fixing bugs in the runtime to harden widely-used sandboxes. It is crucial to clear up this dispute and make expectations clear on both sides, to protect millions of users relying on these sandboxes.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback and Snyk for assisting us with the vulnerability disclosure. We also thank Brian Warner from Agoric and the authors of SandTrap [14], in particular, Daniel Hedin and Mohammad M. Ahmadpanah, for their extensive feedback. This work was conducted in the scope of a dissertation at the Saarbrücken Graduate School of Computer Science.

References

[1] ADsafe. https://github.com/douglascrockford/ADsafe.

[2] box.js. https://github.com/CapacitorSet/box-js.

[3] The Caja Compiler. https://developers.google.com/caja.

[4] embark-code-runner. https://github.com/embarklabs/embark/tree/master/packages/core/code-runner.

[5] Escaping JavaScript sandboxes with parsing issues. https://portswigger.net/research/escaping-javascript-sandboxes-with-parsing-issues. Accessed: 2023-02-27.

[6] How workers works. https://developers.cloudflare.com/workers/learning/how-workers-works.

[7] Mods - user installed extensions. https://github.com/Moddable-OpenSource/moddable/blob/public/documentation/xs/mods.md.

[8] Ses. https://github.com/endojs/endo/tree/master/packages/ses.

[9] Who is using isolated-vm. https://www.npmjs.com/package/isolated-vm#who-is-using-isolated-vm.

[10] Yet another Google Caja bypasses hat-trick. https://blog.bentkowski.info/2017/11/yet-another-google-caja-bypasses-hat.html. Accessed: 2023-02-27.

[11] ABATE, C., DE AMORIM, A. A., BLANCO, R., EVANS, A. N., FACHINI, G., HRITCU, C., LAURENT, T., PIERCE, B. C., STRONATI, M., THIBAULT, J., AND TOLMACH, A. When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise. In *Conference on Computer and Communications Security (CCS)* (2018).

[12] ABDALKAREEM, R., NOURRY, O., WEHAIBI, S., MUJAHID, S., AND SHIHAB, E. Why do developers use trivial packages? an empirical case study on npm. In *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)* (2017).

[13] AGTEN, P., VAN ACKER, S., BRONDSEMA, Y., PHUNG, P. H., DESMET, L., AND PIESSENS, F. JSand: complete client-side sandboxing of third-party JavaScript without browser modifications. In *Annual Computer Security Applications Conference (ACSAC)* (2012).

[14] AHMADPANAH, M. M., HEDIN, D., BALLIU, M., OLSSON, L. E., AND SABELFELD, A. SandTrap: Securing JavaScript-driven Trigger-Action Platforms. In *USENIX Security Symposium* (2021).

[15] ASCHERMANN, C., FRASSETTO, T., HOLZ, T., JAUERNIG, P., SADEGHI, A.-R., AND TEUCHERT, D. NAUTILUS: Fishing for Deep Bugs with Grammars. In *Network and Distributed System Security Symposium (NDSS)* (2019).

[16] BHARGAVAN, K., DELIGNAT-LAVAUD, A., AND MAFFEIS, S. Language-based defenses against untrusted browser origins. In *USENIX Security Symposium* (2013).

[17] BHUIYAN, M., PARTHASARATHY, A. S., VASILAKIS, N., PRADEL, M., AND STAICU, C.-A. SecBench.js: An executable security benchmark suite for server-side JavaScript. In *International Conference on Software Engineering (ICSE)* (2023).

[18] BROWN, F., NARAYAN, S., WAHBY, R. S., ENGLER, D. R., JHALA, R., AND STEFAN, D. Finding and preventing bugs in javascript bindings. In *Symposium on Security and Privacy (S&P)* (2017).

[19] CUTSEM, T. V., AND MILLER, M. S. Proxies: design principles for robust object-oriented intercession apis. In *Symposium on Dynamic Languages (DLS)* (2010).

[20] DAVIS, J. C., COGHLAN, C. A., SERVANT, F., AND LEE, D. The impact of regular expression denial of service (ReDoS) in practice: an empirical study at the ecosystem scale. In *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)* (2018).

[21] DAVIS, J. C., WILLIAMSON, E. R., AND LEE, D. A sense of time for javascript and node.js: First-class timeouts as a cure for event handler poisoning. In *USENIX Security Symposium* (2018).

[22] DONG, X., TRAN, M., LIANG, Z., AND JIANG, X. AdSentry: comprehensive and flexible confinement of JavaScript-based advertisements. In *Annual Computer Security Applications Conference, (ACSAC)* (2011).

[23] DUAN, R., ALRAWI, O., KASTURI, R. P., ELDER, R., SALTAFORMAGGIO, B., AND LEE, W. Towards measuring supply chain attacks on package managers for interpreted languages. In *Network and Distributed System Security Symposium (NDSS)* (2021).

[24] FERREIRA, G., JIA, L., SUNSHINE, J., AND KÄSTNER, C. Containing malicious package updates in npm with a lightweight permission system. In *International Conference on Software Engineering (ICSE)* (2021).

[25] FINIFTER, M., WEINBERGER, J., AND BARTH, A. Preventing capability leaks in secure javascript subsets. In *Network and Distributed System Security Symposium (NDSS)* (2010).

[26] FRASSETTO, T., JAUERNIG, P., LIEBCHEN, C., AND SADEGHI, A. IMIX: in-process memory isolation extension. In *USENIX Security Symposium* (2018).

[27] GAUTHIER, F., HASSANSHAHI, B., AND JORDAN, A. AFFOGATO: runtime detection of injection attacks for node.js. In *International Symposium on Software Testing and Analysis (ISSTA)* (2018).

[28] GHOSN, A., KOGIAS, M., AND PAYER, M. Enclosure: Language-Based Restriction of Untrusted Libraries. In *International Conference on Architectural Support for Programming Languages and Operating Systems, (ASPLOS)* (2021).

[29] GROEF, W. D., MASSACCI, F., AND PIESSENS, F. Nodesentry: least-privilege library integration for server-side javascript. In *Proceedings of the 30th Annual Computer Security Applications Conference, (ACSAC)* (2014).

https://t.me/learningnets

[30] HAN, H., OH, D., AND CHA, S. K. CodeAlchemist: Semantics-aware code generation to find vulnerabilities in JavaScript engines. In *Network and Distributed System Security Symposium (NDSS)* (2019).

[31] HOLLER, C., HERZIG, K., AND ZELLER, A. Fuzzing with Code Fragments. In *USENIX Security Symposium* (2012).

[32] INGRAM, L., AND WALFISH, M. Treehouse: JavaScript sandboxes to help web developers help themselves. In *USENIX Annual Technical Conference (ATC)* (2012).

[33] KEIL, M., AND THIEMANN, P. Transaction-based sandboxing for JavaScript. *CoRR abs/1612.00669* (2016).

[34] KO, Y., REZK, T., AND SERRANO, M. SecureJS compiler: portable memory isolation in JavaScript. In *Symposium on Applied Computing (SAC)* (2021).

[35] KOCHER, P., HORN, J., FOGH, A., GENKIN, D., GRUSS, D., HAAS, W., HAMBURG, M., LIPP, M., MANGARD, S., PRESCHER, T., SCHWARZ, M., AND YAROM, Y. Spectre attacks: Exploiting speculative execution. In *Symposium on Security and Privacy (S&P)* (2019).

[36] KOISHYBAYEV, I., AND KAPRAVELOS, A. Mininode: Reducing the attack surface of node.js applications. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)* (2020).

[37] LEE, S., HAN, H., CHA, S. K., AND SON, S. School of Computing, KAIST. In *USENIX Security Symposium* (2020).

[38] LEHMANN, D., KINDER, J., AND PRADEL, M. Everything old is new again: Binary security of WebAssembly. In *USENIX Security Symposium* (2020).

[39] LI, S., KANG, M., HOU, J., AND CAO, Y. Detecting Node.js prototype pollution vulnerabilities via object lookup analysis. In *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)* (2021).

[40] LI, S., KANG, M., HOU, J., AND CAO, Y. Detecting Node.js prototype pollution vulnerabilities via object lookup analysis. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)* (2021).

[41] LI, S., KANG, M., HOU, J., AND CAO, Y. Mining Node.js vulnerabilities via object dependence graph and query. In *USENIX Security Symposium* (2022).

[42] LIU, Y., ZHANG, M., AND MENG, W. Revealer: Detecting and exploiting regular expression denial-of-service vulnerabilities. In *Symposium on Security and Privacy (S&P)* (2021).

[43] MAFFEIS, S., MITCHELL, J. C., AND TALY, A. Isolating JavaScript with filters, rewriting, and wrappers. In *European Symposium on Researchin Computer Security (ESORICS)* (2009).

[44] MAFFEIS, S., MITCHELL, J. C., AND TALY, A. Object capabilities and isolation of untrusted web applications. In *Symposium on Security and Privacy (S&P)* (2010).

[45] MAFFEIS, S., AND TALY, A. Language-based isolation of untrusted JavaScript. In *Computer Security Foundations Symposium (CSF)* (2009).

[46] MAISURADZE, G., BACKES, M., AND ROSSOW, C. Dachshund: Digging for and securing (non-)blinded constants in JIT code. In *Network and Distributed System Security Symposium (NDSS)* (2017).

[47] MILLER, M. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. Johns Hopkins University, 2006.

[48] NARAYAN, S., DISSELKOEN, C., GARFINKEL, T., FROYD, N., RAHM, E., LERNER, S., SHACHAM, H., AND STEFAN, D. Retrofitting fine grain isolation in the Firefox renderer. In *USENIX Security Symposium* (2020).

[49] NIELSEN, B. B., TORP, M. T., AND MØLLER, A. Modular call graph construction for security scanning of node.js applications. In *International Symposium on Software Testing and Analysis (ISSTA)* (2021).

[50] PARK, S., XU, W., YUN, I., JANG, D., AND KIM, T. Fuzzing JavaScript Engines with Aspect-preserving Mutation. In *Symposium on Security and Privacy (S&P)* (2020).

[51] PARK, T., DHONDT, K., GENS, D., NA, Y., VOLCKAERT, S., AND FRANZ, M. Nojitsu: Locking down javascript engines. In *Network and Distributed System Security Symposium (NDSS)* (2020).

[52] PASHCHENKO, I., VU, D. L., AND MASSACCI, F. A qualitative study of dependency management and its security implications. In *Conference on Computer and Communications Security (CCS)* (2020).

[53] PATRA, J., AND PRADEL, M. Learning to Fuzz: Application-Independent Fuzz Testing with Probabilistic, Generative Models of Input Data. *TU Darmstadt, Department of Computer Science, Tech. Rep. TUD-CS-2016-14664* (2016).

[54] POLITZ, J. G., ELIOPOULOS, S., GUHA, A., AND KRISHNAMURTHI, S. ADsafety: Type-Based Verification of JavaScript Sandboxing. In *USENIX Security Symposium* (2011).

[55] ROMANO, A., LEHMANN, D., PRADEL, M., AND WANG, W. Wobfuscator: Obfuscating javascript malware via opportunistic translation to webassembly.

[56] SEN, K., KALASAPUR, S., BRUTCH, T. G., AND GIBBS, S. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)* (2013).

[57] SHCHERBAKOV, M., BALLIU, M., AND STAICU, C.-A. Silent spring: Prototype pollution leads to remote code execution in Node.js. In *USENIX Security Symposium* (2023).

[58] STAICU, C., AND PRADEL, M. Freezing the web: A study of ReDoS vulnerabilities in JavaScript-based web servers. In *USENIX Security Symposium* (2018).

[59] STAICU, C., PRADEL, M., AND LIVSHITS, B. SYNODE: understanding and automatically preventing injection attacks on NODE.JS. In *Network and Distributed System Security Symposium (NDSS)* (2018).

[60] STAICU, C., TORP, M. T., SCHÄFER, M., MØLLER, A., AND PRADEL, M. Extracting taint specifications for javascript libraries. In *International Conference on Software Engineering (ICSE)* (2020).

[61] STAICU, C.-A., RAHAMAN, S., KISS, Á., AND BACKES, M. Bilingual problems: Studying the security risks incurred by native extensions in scripting languages. In *USENIX Security Symposium* (2023).

[62] TALY, A., ERLINGSSON, Ú., MITCHELL, J. C., MILLER, M. S., AND NAGRA, J. Automated Analysis of Security-Critical JavaScript APIs. In *Symposium on Security and Privacy (S&P)* (2011).

[63] TRAN, T., PELIZZI, R., AND SEKAR, R. JaTE: Transparent and efficient JavaScript confinement. In *Annual Computer Security Applications Conference, (ACSAC)* (2015).

[64] VAHLDIEK-OBERWAGNER, A., ELNIKETY, E., DUARTE, N. O., SAMMLER, M., DRUSCHEL, P., AND GARG, D. ERIM: secure, efficient in-process isolation with protection keys (MPK). In *USENIX Security Symposium* (2019).

[65] VASILAKIS, N., KAREL, B., ROESSLER, N., DAUTENHAHN, N., DEHON, A., AND SMITH, J. M. BreakApp: Automated, Flexible Application Compartmentalization. In *Network and Distributed System Security Symposium (NDSS)* (2018).

[66] VASILAKIS, N., STAICU, C.-A., NTOUSAKIS, G., KALLAS, K., KAREL, B., DEHON, A., AND PRADEL, M. Mir: Automated Quantifiable Privilege Reduction Against Dynamic Library Compromise in JavaScript. In *Conference on Computer and Communications Security (CCS)* (2021).

[67] VEGGALAM, S., RAWAT, S., HALLER, I., AND BOS, H. Ifuzzer: An evolutionary interpreter fuzzer using genetic programming. In *European Symposium on Research in Computer Security (ESORICS)* (2016).

https://t.me/learningnets

[68] WYSS, E., WITTMAN, A., DAVIDSON, D., AND CARLI, L. D. Wolf at the door: Preventing install-time attacks in npm with latch. In *Asia Conference on Computer and Communications Security (ASIA CCS)* (2022).

[69] XIAO, F., HUANG, J., XIONG, Y., YANG, G., HU, H., GU, G., AND LEE, W. Abusing hidden properties to attack the Node.js ecosystem. In *USENIX Security Symposium* (2021).

[70] YE, G., TANG, Z., TAN, S. H., HUANG, S., FANG, D., SUN, X., BIAN, L., WANG, H., AND WANG, Z. Automated conformance testing for javascript engines via deep compiler fuzzing. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021* (2021).

[71] YEE, B., SEHR, D., DARDYK, G., CHEN, J. B., MUTH, R., ORMANDY, T., OKASAKA, S., NARULA, N., AND FULLAGAR, N. Native client: A sandbox for portable, untrusted x86 native code. In *Symposium on Security and Privacy (S&P)* (2009).

[72] ZELLER, A. Isolating cause-effect chains from computer programs. In *Symposium on Foundations of Software Engineering (FSE)* (2002).

[73] ZHANG, M., AND MENG, W. JSISOLATE: lightweight in-browser JavaScript isolation. In *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)* (2021).

[74] ZIMMERMANN, M., STAICU, C., TENNY, C., AND PRADEL, M. Small world with high risks: A study of security threats in the npm ecosystem. In *USENIX Security Symposium* (2019).

Sandbox	Vulnerability
`vm2`	Issues 179
`vm2`	Issues 186
`vm2`	Issue 197
`vm2`	Issue 199
`vm2`	Issue 224
`vm2`	Issue 225
`vm2`	Issue 241
`vm2`	Issue 268
`vm2`	Issue 276
`vm2`	Issue 187
`vm2`	Issue 185
`vm2`	Issue 184
`vm2`	Issue 177
`vm2`	Issue 175
`vm2`	Issue 138
`realms-shim`	Advisory GHSA-7cg8-pq9v-x98q
`realms-shim`	Advisory GHSA-6jg8-7333-554w
`safe-eval`	Issue 19
`safe-eval`	Issue 18
`safe-eval`	Issue 16
`safe-eval`	Issue 24
`safe-eval`	Issue 7
`safe-eval`	Issue 5
`ADSafe`	Politz et al. [54]
`Caja`	Michał Bentkowski's blog post [10]
`Caja`	PortSwigger blog post [5]

Table 5: Complete list of vulnerabilities depicted in Table 1.

A Complete list of studied vulnerabilities

To allow further analysis and/or replication of our empirical study, we present in Table 5 the complete list of vulnerabilities considered in Table 1.

https://t.me/learningnets