SecBench.js: An Executable Security Benchmark Suite for Server-Side JavaScript

Masudul H. M. Bhuiyan
CISPA Helmholtz Center for Information Security
masudul.bhuiyan@cispa.de

Adithya Srinivas Parthasarathy
Indian Institute of Information Technology, Design, and Manufacturing
adithyasrinivas11@gmail.com

Nikos Vasilakis
Massachusetts Institute of Technology
nikos@vasilak.is

Michael Pradel
University of Stuttgart
michael@binaervarianz.de

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

Abstract

Npm is the largest software ecosystem in the world, offering millions of free, reusable packages. In recent years, various approaches for protecting this ecosystem have been developed, e.g., techniques for identifying vulnerabilities and for defending against attacks at runtime. However, we notice the lack of a comprehensive and widely used benchmark for npm security. Instead, every new paper is evaluated on a different, hand-picked sets of vulnerabilities. The lack of a benchmark not only repeatedly forces researchers to develop an evaluation dataset in addition to their prototype, but also makes it difficult to compare different techniques with each other. This paper presents SecBench.js, a comprehensive benchmark suite of vulnerabilities and executable exploits for npm. The benchmark comprises 600 vulnerabilities, each with a payload that exercises the problem and an oracle that validates successful exploitation. Moreover, when available, we also provide additional metadata for each vulnerability, such as the location of the vulnerability and a pointer to the fixed version of the package. Our benchmark suite covers the five most common vulnerability classes for server-side JavaScript. As applications of SecBench.js, we perform several studies that are directly enabled by our suite. For example, cross-checking SecBench.js against existing security advisories reveals 168 vulnerable versions in 19 packages that are mislabeled in the advisories. As another example, we show that applying simple code transformations to the exploits in our suite helps identify flawed fixes of vulnerabilities. In total, we identify 20 zero-day vulnerabilities, most of which were already acknowledged by practitioners.

1 Introduction

JavaScript is one of the most popular languages, and its continuous growth is fueled by the npm ecosystem, a repository with almost two million reusable packages. Vulnerabilities are common in npm and pose a major threat to applications and the ecosystem as a whole. There are at least 5,500 security advisories issued for this ecosystem, the result of several years of efforts invested both by practitioners and academics. Due to the reusable nature of these packages, this domain is increasingly known as *software supply chain security*, not to be confused with a subclass of problems in this area, called supply chain attacks.

There is a large body of recent research work on software supply chain security, targeting the npm ecosystem, and server-side JavaScript in particular. First, there are papers studying specific classes of security problems: injection [21, 45], ReDoS [12, 44], prototype pollution [30], hidden property attacks [53], path traversal [23], supply chain attacks [18, 48, 57], outdated [29, 38, 57] or trivial [1] dependencies, bugs in low-level code [8]. Second, there is work on developing new defenses and detection techniques using static [21, 30, 31, 35, 36, 48, 52], dynamic [11, 13, 14, 28, 50, 51], or hybrid [18, 45] analysis techniques.

Unfortunately, the software security community lacks a rigorous collection of server-side JavaScript vulnerabilities that serves as a benchmark for evaluating software security research. In contrast to other communities, such as databases [40], parallel processing [4], graphics [47], and machine learning [10], where established benchmark suites are used routinely to assess and demonstrate the relative merits of different contributions, security research often relies on ad hoc sets of programs collected manually by the authors of the respective papers. While benchmark sets are not without limitations, the lack of a security-oriented benchmark suite for npm has important implications: (1) it slows down research, as authors of each paper have to look for a new set of benchmarks that satisfies their needs and may convince reviewers, (2) it obfuscates the true merits of different proposals, as they are not compared across a single set of vulnerabilities, and (3) it complicates the evaluation process, as different audiences focus on their own corner of the space of all vulnerabilities rather than a unified rubric. A widely-used security-oriented benchmark suite for npm would alleviate these problems, and help improve experimental techniques and metrics in this area.

Requirements for a security benchmark suite: The goal of our work is to define a benchmark suite for server-side JavaScript that can be used to design and evaluate next-generation defenses against software system threats. Therefore, we opt for a security benchmark suite that is:

- *Realistic*: We aim at a benchmark suite built from a diverse set of real-world software, with as few modifications of them as possible. Such realism ensures that success on the benchmark is likely to generalize to other real-world security problems.
- *Executable*: We want the suite to include inputs that trigger the vulnerabilities and a runtime oracle that checks whether an attempt to exploit a vulnerability indeed triggers the effect anticipated by the attacker. Having such executable exploits serves as a signal that a vulnerability can be exploited, enables evaluating runtime detection and mitigation techniques, and allows for studying the runtime properties of vulnerabilities.
- *Two-sided*: We aim at a benchmark that includes both vulnerable and fixed versions of the code. Providing both sides is important for measuring the false positive rate of detection and mitigation techniques, but also for studying how vulnerabilities get fixed.
- *Vetted*: We study each vulnerability in the benchmark suite to confirm its existence, ensure that it falls under a certain attack class, and generate appropriate metadata. This requirement is in contrast to very large-scale, automatically gathered datasets, which are used, e.g., to train deep learning-based vulnerability detectors, and typically suffer from some noise.

Despite the increasing importance of server-side JavaScript security, there currently is no benchmark of vulnerabilities that matches all four of the above criteria. Table 1 summarizes the most closely related existing benchmarks and datasets. One group of them consists of synthetic vulnerabilities, either created manually [9], via template-based code generation [7], or automatically by mutating existing code [17]. These benchmarks lack realism and hence do not suit our needs. Another group consists of automatically gathered datasets, e.g., created based on commits that mention a vulnerability [19, 20]. These datasets do not come with executable exploits and lack manual vetting, which causes noise, e.g., in the form of unrelated code changes tangled with a commit. Finally, the third group of benchmarks consists of manually curated vulnerabilities [25, 39], but lack executable exploits that use the vulnerabilities to trigger a security-relevant action. For example, MAGMA [25] provides inputs that show that vulnerabilities can cause a crash, but not that they can be further exploited by an attacker.

The SecBench.js benchmark suite: This paper presents SecBench.js, the first benchmark suite of JavaScript vulnerabilities that fulfills all four of the above requirements.

Table 1. Comparison with existing vulnerability benchmarks and datasets.

Suite	Language	Vulnerabilities	Realism	Executable exploits	Two-sided	Vetted
CGC [9]	C	590	✗	✓	✗	✓
Juliet [7]	C/C++, Java, C#	121,922	✗	✓	✓	✓
LAVA-M [17]	C	2,265	✗	✓	✓	✗
BigVul [19]	C/C++	3,745	✓	✗	✓	✗
Ferenc et al. [20]	JavaScript	1,496	✓	✗	✓	✗
VulinOSS [22]	various	17,738	✓	✗	✗	✗
Magma [25]	C	118	✓	✗	✓	✓
Ghera [34]	Java/Android	25	✓	✓	✗	✓
Ponta et al. [39]	Java	624	✓	✗	✓	✓
SecBench.js	JavaScript	600	✓	✓	✓	✓

The benchmark consists of 600 publicly reported vulnerabilities contained in widely used advisory databases, such as Snyk and GitHub. SecBench.js spans a diverse set of five threat classes, namely the five most common threat classes for benign server-side JavaScript: path traversal, prototype pollution, command injection, denial-of-service, and code injection. Each vulnerability comes with an executable exploit, i.e., an attack input that triggers the vulnerability and causes behavior unexpected by the vulnerable software package. Additionally, we also provide test oracles to judge the success of each exploit. The benchmark is two-sided: if available, it includes a fix of the vulnerability where the provided exploit often fails. Finally, the benchmark is vetted, as we manually validate each vulnerability. The full benchmark suite is open-sourced, well-documented, and packaged conveniently in a container image.

Empirical study: To showcase the usefulness of SecBench.js, we perform several studies that would either be costly to carry out or error-prone without our suite:

- *Sink extraction*: Using dynamic analysis we automatically extract vulnerability location for 94% of the entries in our suite, saving important annotation time, and reducing the likelihood of human-introduced errors. Through manual analysis, we confirm that the extracted information is in line with human expectations.
- *Dynamic behavior*: For each vulnerability, we study the percentage of the API used by the exploit and the maximum length of call chains observed at runtime. We find that most entries have a short sequence of

calls, i.e., ≤ 5, and that the exposed API is either very small or very large. We discuss the implications of these findings for future designs of program analyses in this domain.

- *Mislabeled versions*: By running our exploits on different versions of the vulnerable packages, we identify 168 wrongly classified versions in 19 packages. In two cases, the latest release of the package is wrongly classified, corresponding to zero-day vulnerabilities.
- *Flawed patches*: We apply four simple code transformations to the exploits in SecBench.js to test the deployed patches against variants of the original attack. We identify 18 flawed patches and, at the time of writing, we were assigned 12 CVEs for these findings.

2 Methodology

In this section, we first discuss our threat model (§2.1) and then present our methodology for building the benchmark suite: collecting known vulnerabilities (§2.2, §2.3), adapting or newly developing a proof-of-concept exploit (§2.4), and collecting additional metadata (§2.5, §2.6).

2.1 Threat Model

We focus on security defects, i.e., vulnerabilities, and exclude malicious packages. There are several reasons for this decision. Malicious packages are often removed from the repositories immediately after detection. Hence, one would need to perform continuous data collection for detecting such packages before deletion. Additionally, we would need to distribute these potentially harmful samples ourselves which would raise ethical and legal concerns. Thus, the methodology for collecting, distributing, and executing malicious samples would be significantly different than the one presented in this paper. While we believe this can be an important research endeavour on its own, in this work we focus on mistakes made by developers, i.e., vulnerabilities.

Most of the considered entries in our suite are libraries. By construction, these software components have incomplete threat models: Since it is not clear where the library's inputs come from, one should assume the worst-case scenario, i.e., that they are attacker-controlled. Our suite does not make any judgment on the feasibility of this threat model, but instead relies on the assessment of practitioners about the risk posed by such vulnerabilities.

We focus on packages that can be executed on the server-side, ignoring typical client-side issues, e.g., cross-site scripting and open redirects. This is because we want to offer a uniform execution environment for our suite, i.e., Node.js, without the need to simulate a browser-like environment. We also assume that the underlying runtime is trusted and free of bugs, i.e., we consider bugs in the source code of Node.js out of scope.

To be representative of the threats that practitioners are most interested in, we focus on the largest classes of vulnerabilities. Specifically, we select five classes of vulnerabilities, which all are targeted by recent work in this domain: command and code injection [21, 31, 35, 45, 51, 51], ReDoS [3, 11–13, 33, 44] , prototype pollution [30, 31], and path traversal [23, 31].

2.2 Source of Vulnerabilities

We gather vulnerabilities from three different sources: Snyk, GitHub Advisories (previously called Npm Advisories), and Huntr.dev. These sources include thousands of vulnerabilities, often accompanied by an exploit, are popular among developers, and were extensively used in previous work [15, 31, 51, 57]. For each considered source, we perform an initial web scrapping of its web interface to collect the number of vulnerabilities and their types.

2.3 Filtering of Candidate Vulnerabilities

We only include in our suite vulnerabilities for which we can present an input that triggers a security-relevant action, as described below. Thus, we exclude vulnerabilities (a) in packages that we cannot install or that were deleted, (b) that we cannot reproduce, (c) that are incompatible with our setup, e.g., operating system.

It is worth mentioning that the security landscape in npm is atypical: while for most other ecosystems we are aware of, vulnerabilities are often published without a proof-of-concept exploit, in npm that is not the case. Hence, our benchmark creation process is simplified by the detailed information available in many of the vulnerability reports. When no exploit is readily available, we try to create one, allocating a budget of one hour per vulnerability to this task.

2.4 Executable Exploits

Inspired by widely-used unit testing practices, each exploit in SecBench.js contains five parts: (i) a setup phase in which the target package is required or started, (ii) a sanity check that the security-relevant post-condition is not met before the actual payload, (iii) an API call that delivers the problematic input to the vulnerability endpoint, (iv) a check that the post-condition is met, and (v) a cleanup phase in which the side-effects of the test are reverted.

The reader may notice the unusual semantics for "passed" tests: a test is considered successful if the vulnerability could be used to trigger the *security-relevant action*. This is in contrast to a test that the maintainer of a package might write.

For example, the security-relevant action for injection vulnerabilities is the creation of a file on disk. As a sanity check, we verify that this file is not yet present on the disk at the beginning of the test. After the payload's execution, we assert the presence of the file. Thus, the payload needs to inject code that loads the file system API and creates the target file.

2.5 Vulnerability's Location

Having precise information about the vulnerability location is important for evaluating static analysis techniques. These analyses often point to a problematic location in the code, which needs to be then judged against a ground truth.

To identify the location of vulnerabilities, we adopt the notion of sinks in taint-style analysis. A sink here is a relevant built-in API that is used to deliver the payload. For example, a typical sink for command injection is `child_process.exec()`. We note that the sink call can be located either in the vulnerable package or in one of its dependencies. For a given package, we define the sink location as the line of code at which the payload is passed into the sink or into a third-party dependency that further propagates it to the sink.

To extract sink locations, we use a simple yet effective dynamic analysis. We run each exploit in a prepared environment in which relevant sink APIs are hooked. For example, for prototype pollution we add a custom setter on the property `polluted` in `Object.prototype`. Each time an exploit sets that property, our analysis code is triggered and we inspect the stack trace to extract the location that triggers the property set. Similarly, we hook `child_process` APIs for command injection, `fs` for path traversal, regular expression matching for ReDoS, and `eval` and `Function` for code injection. In case of multiple sink calls, we only extract the first sink location.

2.6 Vulnerability's Patch

Including the information about the patched version and the fixing commit in our suite is important for enabling studies of deployed fixes, as discussed in Section 4.3. Moreover, having fixed versions of the vulnerable code is essential for judging the precision of static vulnerability detection tools. Such tools should detect a problem in the vulnerable version, but not in the fixed one.

We follow two strategies for extracting the fixes for a vulnerability. First, we scrape the considered sources to extract this information, but we notice that in some cases these sources are incomplete. Hence, we also run our payloads on the latest version of the package and check if it still works. In case the exploit does not, but the considered sources claim there is no patch available, we first analyze the error logs for any relevant clues that a patch is indeed deployed. We then proceed to analyze the vulnerability location to detect the presence of a patch. If we confirm that there is a fix in place, we analyze the repository's history to identify the fixing commit.

3 The SecBench.js Benchmark Suite

In this section, we describe our benchmark suite as a reusable research artifact: we quantify its most important components and present details about its implementation and structure.

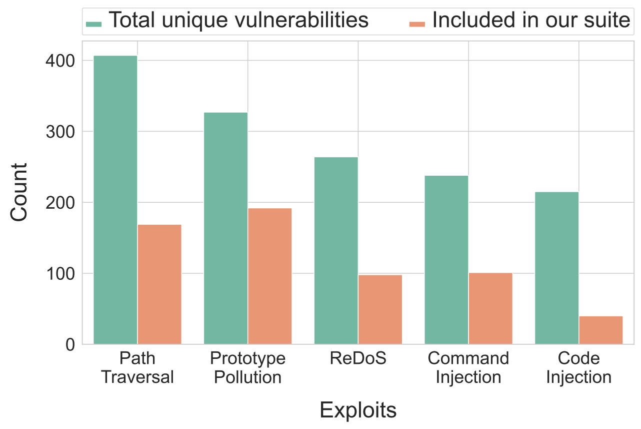

Figure 1. Classes of vulnerabilities included in SecBench.js.

3.1 Descriptive Statistics

In Table 1, we show the five classes of vulnerabilities considered by our threat model (§2.1). The suite cover 34.44% of all vulnerabilities in these classes. In total, we consider 577 vulnerable packages, representing 23.43% of all the packages with known vulnerabilities.

Table 3 shows the number of exploits in SecBench.js, grouped by vulnerability type, and the number of their metadata entries. For 48% of the vulnerabilities, we provide information about the fixed version and the fix commit. In particular, for nine entries, this information was not included in the corresponding security advisory, but we identified it using the second approach described in Section 2.6. For 94% of all vulnerabilities, we provide information about the vulnerability location (sink), extracted using the approach described in Section 2.5. We manually sampled 45 such sinks and judged their correctness. We identified only two cases for which our analysis produced a wrong sink, and we proceeded to fix the corresponding bug in the analysis. Finally, we provide CVE information for 67% of the exploits in our suite. We note that the lack of metadata for some entries is not necessarily a shortcoming of our methodology, but a reflection of the underlying data, i.e., there are several vulnerabilities for which no CVE was assigned, or no patch deployed.

384 of the exploits in SecBench.js correspond to vulnerabilities reported both on Snyk and GitHub Security Advisories, while 203 and 7 were exclusively reported on Snyk and GitHub, respectively. For each vulnerability, our metadata links to the corresponding advisories in these databases.

For most of the exploits in SecBench.js, we start with an exploit fragment, i.e., either a working PoC or a description in natural language of the exploit. Hence, most of the effort went into encoding this knowledge in a uniform way, adapting it to our framework, and adding oracle checks. As a measure of the suite's development effort, we report the number of assertions in our tests, which were all manually written by us: SecBench.js has a total of 1,244 assertions,

Table 2. Number of exploits included in SecBench.js, together with number of metadata entries, for each considered vulnerability type.

Type of vulnerability	Number of exploits	Fixed version	Sink location	CVE Information
Path Traversal	169	19	161	80
Prototype pollution	192	126	186	158
ReDoS	98	78	89	59
Command Injection	101	41	99	90
Code Injection	40	21	32	20
Total	600	285	567	407

```
1  {
2    "id": "CVE-2019-10744",
3    "dependencies": {
4      "lodash":"4.17.10"
5    },
6    "links": {
7      "source1":"SNYK-JS-LODASH-450202",
8      "source2":"GHSA-jf85-cpcp-j695"
9    },
10   "fixedVersion":"4.17.12",
11   "fixCommit":"
         c3fd203b3be87a8177f7f00824033c95f981f984",
12   "sinkLocation": "lodash.js:2573:21"
13 }
```

Figure 2. Metadata corresponding to the entry for lodash's vulnerability CVE-2019-10744.

thus, an average of 2.07 assertions per test. Additionally, for most tests, we have a meta assertion on the number of oracle checks that must be performed during the execution. We found this to be useful for ensuring that all the asynchronous tasks were correctly executed.

On average, each vulnerable package directly depends on 1.42 other packages, and is depended upon by 947.62 packages. This shows that the considered packages are relatively influential and that there are important interactions with third-party packages.

3.2 Implementation of the Benchmark Suite

When implementing SecBench.js, we rely whenever possible on industrial-grade developers tools that significantly reduce our development cost, and at the same time provide a robust scaffolding for the suite. Moreover, we believe that users of SecBench.js are already familiar with these technologies, thus leading to an easier adoption. Finally, these tried-and-tested building blocks enable us to efficiently run the suite and easily integrate analysis code at runtime.

At a high level, SecBench.js is a set of unit tests, where each successful exploit is considered a passing test. We use the Jest[1] testing framework for writing the tests.

SecBench.js relies on the package managers such as `npm` or `yarn` to distribute the vulnerable package versions. Each vulnerability is contained in a separate folder, containing a test file `foo.test.js` with the actual inputs and oracle checks, and a metadata file `package.json` with information about the vulnerable version, the fix, the exact location, and external resources.

For example, in Figure 2 we show our metadata entry for the vulnerability CVE-2019-10744. Clients of the suite can install the vulnerable version `4.17.10`, e.g., by running `npm install` in the folder containing these files. We pinned the exact vulnerable version in the metadata file, preventing the package manager from installing a newer version of the vulnerable package. Nonetheless, clients can install the fixed version or any other version of the target package by changing the entry at line 4 or by instructing the package manager directly to install a different version, e.g., `npm install lodash@4.17.12`.

To simplify batch installation, SecBench.js organizes its metadata and entries in a hierarchical structure. At first, we aggregate the individual vulnerability folders by vulnerability class, offering an aggregated `package.json` file that refers to all the individual vulnerabilities. For example, all the ReDoS vulnerability are comprised in the `redos` folder, containing a `package.json` with 98 internal dependencies. Running the package manager in that folder will download all the 98 vulnerabilities of that class, together with their dependencies. Similarly, SecBench.js also allows for batch installing all the 600 vulnerable packages, by providing a main `package.json` that refers to the individual ones corresponding to the five vulnerability classes considered in the benchmark. In our setup, it takes twelve minutes to run `npm install` in the main folder of SecBench.js, i.e., installing all the 600 vulnerable packages, with their exploits.

The presented folder-based structure allows us to have multiple vulnerable versions for the same package. For example, for `lodash`, SecBench.js includes four different prototype pollution vulnerabilities and one ReDoS vulnerability.

[1] https://jestjs.io/

```
test("prototype pollution in lodash", () => {
  // setup
  const mergeF = require("lodash").defaultsDeep;
  const payload = '{"constructor": {"prototype":
       {"polluted": "yes"}}}';
  // sanity check
  expect({}.polluted).toBe(undefined);
  // exploit
  mergeF({}, JSON.parse(payload));
  // oracle check
  expect({}.polluted).toBe("yes");
  // cleanup
  delete Object.prototype.polluted;
});
```

Figure 3. Example entry in our benchmark suite for `lodash`'s vulnerability CVE-2019-10744.

In Figure 3, we show the test corresponding to the metadata entry discussed earlier. We first set up the test by requiring the vulnerable package and initializing relevant constants. We then assert that the target property is not inadvertently present in the global scope. After that, we trigger the payload by passing it to the vulnerable module. Finally, we assess the presence of the target property and hence, the payload's success. This test can be run using `jest`'s command line interface.

We also note that some tests in SecBench.js have more complex setup and teardown phases. For example, for path traversal vulnerabilities we often need to start the target package in a separate process to act as a web server, perform a HTTP request, and finally, stop the server. Nonetheless, the high-level structure of all our tests is similar.

Considering the hierarchical folder structure discussed earlier, and the test resolution algorithm of `jest`, one can run multiple tests at once by invoking the command line tool with the desired folder as the argument. For example, to run all the ReDoS tests one can run, `jest ./redos` in the root folder of SecBench.js. By default, `jest` runs all the tests in parallel, achieving the following test execution times on a standard consumer laptop: 156s, 13s, 57s, 518s, and 12s for ReDoS, command injection, code injection, path traversal, and prototype pollution, respectively. The two slowest classes are ReDoS, for which the payloads trigger a significant slowdown in the target packages, and path traversal, for which we run the tests sequentially because multiple of them try to serve requests on the same HTTP port. Nonetheless, these results show that users of the suite easily run the tests multiple times per day.

Finally, we also distribute a Docker container containing a ready-to-use version of our suite. This is because some of the tests require certain native libraries to be present on the machine before installation. Our container runs Ubuntu 18.04 and installs a total of seven prerequisites.

3.3 Implementation of Oracles

Testing oracles are an important mechanism in SecBench.js that allow us to judge the outcome of each test. We implement them in the form of assertions on the system, runtime, or program state after a test execution. The oracles ensure that every considered vulnerability can be exploited for performing a security-relevant action. That is, we do not include in our suite vulnerabilities for which we could not provide an input that triggers such an action.

An important observation is that oracles and actions are specific to a given class of vulnerabilities. For example, one can use a ReDoS vulnerability to trigger a slow computation in the main thread, or a prototype pollution to pollute the global scope. However, since these undesired actions are very different in nature, it is only natural that the oracles that assert their success are also different. Below, we discuss in detail each type of oracle used by SecBench.js.

- *Prototype pollution:* The oracle checks the existence of a variable called `polluted` in the global scope. At the beginning of a test, the oracle checks that such a value is not present, and once the payload has been executed, it asserts its existence. We note that this is a very strict oracle and that there are prototype pollution attack vectors that do not allow such an action to be performed, e.g., because they only allow the addition of properties on specific built-in APIs like `Function.prototype`. Nonetheless, since the security impact of such pollutions is limited, we decided to opt for a powerful oracle that only asserts the most serious version of these attacks.
- *Path traversal:* SecBench.js ships with a file called `flag.txt`, which the path traversal vulnerabilities aim to read. The oracle checks that the content served by the vulnerable npm package is the same as the one in this file. As the absolute path to this file varies from machine to machine, hence SecBench.js dynamically adjusts the payload to point to the indicated file.
- *Injections:* The oracle checks the existence of a custom-named file on the disk. We note that for most exploits, this implies loading the built-in `fs` module (code injections) or the `touch` UNIX utility (command injection) to perform the file creation. At the beginning of a test, the oracle checks that the file is not present on the disk, and after the payload is executed, it asserts its existence.
- *ReDoS:* The oracle checks that the target payload takes more than one second to execute. While this simple oracle may lead to both false negatives and false positives, we minimize their likelihood by (i) dimensioning our exploits so that in our setup the computation takes significantly longer than one second, and (ii) the one-second threshold is large enough so that benign computations rarely trigger so long operations.

3.4 Deploying Analysis Code

An important use case we envision for our suite is the development of runtime analyses or defenses. To this end, one needs to deploy analysis code that runs along with our exploits. We detail below, two ways in which analysis code can be injected.

First, due to the integration between Jest and Babel[2], an off-the-shelf transpiler for JavaScript, one can deploy runtime instrumentation using Babel. To do so, one needs to develop a custom Babel transformation and specify it in `babel.config.js`, so that the test environment transforms all the code loaded at runtime using this transformation. In Section 4.1, we show examples of using this technique.

Second, a widely-used dynamic analysis technique for scripting languages is monkey patching, i.e., augmenting the semantics of built-in APIs to include analysis code. By leveraging Jest's setup phase, we can define custom analysis files that are executed before each test and can thus alter the test's global scope. To output the results of a monkey-patched API, one can use custom Jest reporters. More specifically, one can define `setupFiles` and `reporters` in a `jest.config.json` to instruct the testing framework to run the corresponding files before and after the test's execution. We successfully use this analysis technique to extract the sink location for the vulnerabilities in our suite.

4 Dynamic Analysis-based Study of Vulnerabilities

The availability of SecBench.js enables studying properties of vulnerabilities via dynamic analysis at a previously impossible scale. The following shows three examples of such studies. We first analyze how complex the interaction with a vulnerable package needs to be to exploit a vulnerability (Section 4.1). We then study how many versions of a given package are affected by our exploits, and whether this information matches the version constraints provided in the advisory (Section 4.2). Finally, we show that applying simple mutations to our exploits helps identify zero-day vulnerabilities in insufficiently patched packages (Section 4.3).

4.1 Complexity of Runtime Behavior of Exploits

To better understand how complex the interaction with a vulnerable package needs to be to exploit a vulnerability, we take two runtime measurements for each exploit.

Setup First, we measure the *maximum call chain length* triggered by the exploit code, i.e., the longest sequence of transitive function invocations, observed at any point during execution. Second, we measure the *API coverage*, i.e., the percentage of all APIs of the vulnerable package that is used by the exploit. We believe that this would shed light on the feasibility of automatic vulnerability detection, using various program analysis techniques. For this experiment, we ignore the path traversal entries, for which we trigger the exploit using HTTP requests, hence the API coverage is hard to quantify for this case. To extract the two metrics, we interpose measurement wrappers on the relevant require statements, and we instrument all the function definitions to log the entrance and exit points in each function.

[2]https://babeljs.io/

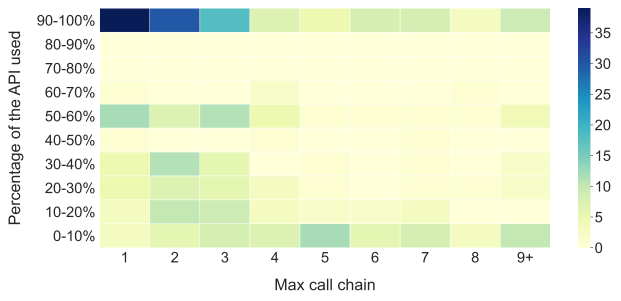

Figure 4. Runtime behavior of the vulnerable packages in SecBench.js. On the y-axis we show the API coverage for our exploits, and on the x-axis the length of the longest call chain observed at runtime. Each square shows how many packages in our suite exhibit that particular runtime behavior.

Results In Figure 4, we show a distribution of runtime complexity for SecBench.js's entries. For example, the exploit in Figure 3 uses less than 1% of the `lodash` API and the maximum call chain length we observe at runtime is 17. Hence, we depict this entry in the bottom right corner of the heat map. A first striking impression regarding Figure 4 is how sparse it is, i.e., most of the data points are clustered around the edges or in the bottom left corner. That is mainly due to disparities on npm, i.e., SecBench.js contains either very small packages, with only a couple of APIs exposed like `dns-sync`, or very large ones like `ramda`.

First, by analyzing the x-axis, we note that the majority of the exploits trigger at most call chains of length three. Nonetheless, there are also complex modules with more than 10 function calls chained at runtime. We argue that vulnerabilities with short call chains are a good candidate for detection with static analyses tools, which tend to struggle with complex inter-procedural flows, for JavaScript.

Second, by analyzing the API coverage (y-axis), we observe that there are many entries covering all the exposed API, others covering 50% of it, and an important cluster using only 10% of the API or less. Since our exploits only trigger one or two methods of the API, we conclude that 100% API coverage often corresponds to trivial packages [1] with a single exposed method, and low coverage to complex ones like `lodash`, exposing tens or hundreds of methods. We argue that for the low-coverage case, purely dynamic analysis techniques would struggle to trigger the vulnerability due to the large API exposed.

Implications To put it all together, we argue that (i) the entries in the upper left corner of Figure 3 are easy to detect with either static or dynamic analysis, (ii) the ones in the upper right corner are a good candidate for dynamic analysis, (iii) the bottom left ones for static analysis, and (iv) the ones in the bottom right are hard to detect with either technique. Since our suite is an aggregation of known security advisories, it may be that the vast majority of previously reported bugs were detected by static analysis techniques, thus the bias towards the left of the heatmap. We argue that future work on JavaScript program analysis should aim to handle first and foremost the entries in the bottom right corner of Figure 3.

4.2 Vulnerable Versions

Since there could be multiple versions of a package affected by the same vulnerability, we hypothesize that one can run our exploits against all versions of the packages to identify affected releases. We then cross-check this information with the original advisories to detect inaccuracies.

Study For every package in SecBench.js, we iteratively update it to every one of its versions and attempt to run the exploit against that particular version, storing the test outcome for each attempt. Nonetheless, running all versions of a package on a single Node.js version is not trivial. Especially legacy versions of a package may not be compatible with our setup, or there may be breaking changes of the used APIs, along the history of the project. That is why we point out that our result may contain false negatives, i.e., a version of a package for which the exploit does not work can still be vulnerable. Finally, we check if all the vulnerable versions we detect are flagged accordingly by the Snyk database.

Results Figure 5 shows the number of vulnerable versions for different packages in our suite. Most of the considered packages have less than 20 versions, in total, while there are a handful of packages with thousand of releases. The mean number of vulnerable versions per package is 19.06, even though 50% of the packages have $\leq$ 5 vulnerable versions, while `@firebase/util` has a maximum of 1,487 vulnerable versions. To better understand this distribution, we plot it in detail in Figure 6. We note that it is very rare the case in which a single version of a package is vulnerable to our exploit, and that there are more than 100 packages with exactly three vulnerable versions.

We envision multiple applications for these prooved-to-be-vulnerable versions. First, they provide natural data augmentation for our suite: instead of 600 vulnerable packages, one may argue that SecBench.js provides 10,922 entries, i.e., counting all the vulnerable versions. Automatic detection techniques should consider analyzing this extended suite for ensuring sensitivity to minor changes in the functionality of vulnerable packages. We believe this is especially important for machine learning-based techniques, where overfitting often occurs. Second, one can cross-check the detected vulnerable versions with the original databases in which the vulnerabilities were reported. Since these knowledge bases are manually curated, this approach may reveal inaccuracies in the original advisories, but also possible regressions or flawed fixes. We study this idea in detail below.

For each package in SecBench.js, we check if there are inconsistencies between the vulnerable versions we detected and the original advisory. We manually check every mismatch and confirm 168 vulnerable versions in 19 packages that are incorrectly flagged as not vulnerable by Snyk. We are in the process of reporting all these inconsistencies with associated examples to Snyk.

In Figure 7, we show examples of mismatches between SecBench.js and the Snyk database. As mentioned earlier, our analysis might produce false negatives, thus, we only focus on cases where our analysis flagged a version as vulnerable but Snyk database has marked that as the opposite. There are three important classes of mismatches we discovered. First, there are cases like `changeset` or `underscore.string` when the constraint provided in the security advisory fails to capture legacy versions, released before the actual interval specified by the constraint. We believe that such cases are low severity as most clients probably do not run such old versions in production. Then, there are cases like `ts-dot-prop` when the constraint fails to capture versions after the indicated interval. Such errors might give users a false sense of security, since audit tools like `snyk-cli` would flag these versions as safe to use. Nonetheless, the affected releases might still be considered as legacy. On the contrary, for two packages, the latest version of the package is incorrectly flagged as not vulnerable, hence, representing a zero-day vulnerabilities. Naturally, these are the most serious issues we discovered, as developers are encouraged to migrate whenever possible to the latest version.

The initial security advisory for `jspdf` identifies a regular expression vulnerable to ReDoS, in version 2.3.0 and earlier:

```
/^data:(\w*\/\w*);*(charset=[\w=-]*)*;*$/
```

The advisory also specify that the problem was fixed in a later version and links to a commit that refines the regular expression:

```
/^data:(\w*\/\w*);*(charset=(?!charset=)[\w=-]*)
    *;*$/
```

The lookahead is supposedly preventing the exponential slow-down problem. Since there was no exploit available for this vulnerability, we developed our own by studying the original vulnerable expression. We were surprised to see that as part of the experiments performed in this section, the exploit also worked against the fixed version, showing that the look ahead patch is ineffective.

When designing this experiment we were aiming to find regressions, i.e., cases in which a given vulnerability was patched, but later the fix was jeopardized by future changes.

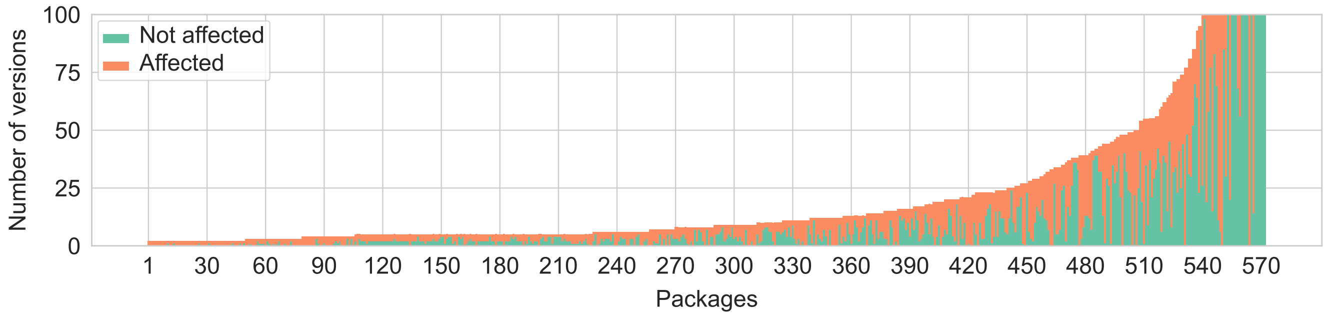

Figure 5. Number of versions per package affected by the exploits in SecBench.js. The packages are sorted in ascending order based on the number of available versions. For readibility, the y-axis is clipped at 100.

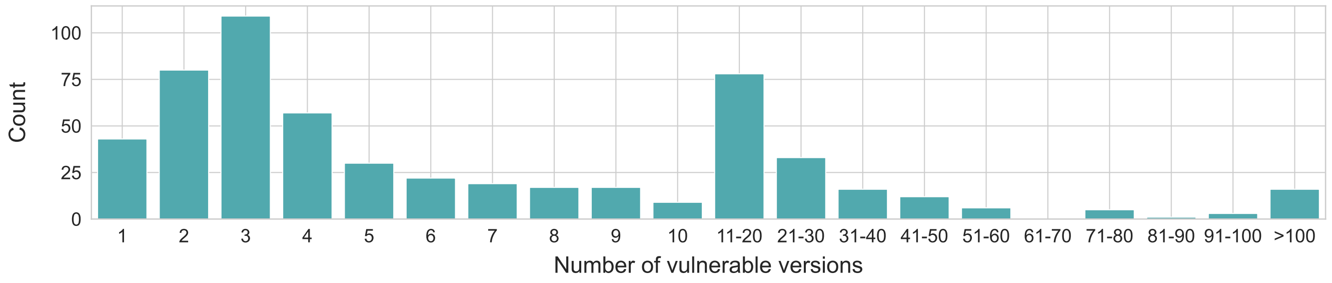

Figure 6. Distribution of number of vulnerable versions for a given package, affected by the exploits in our suite.

Such cases were extensively documented in the software engineering literature [6]. Since we did not encounter such cases in practice, we hypothesize that security patches do not exhibit this behavior, i.e., developers are more careful not to alter checks put in place for security than they are with fixes for other types of bugs. In the process of obtaining this negative result, we encountered a difficult problem that we believe is the main root cause for the inconsistencies described earlier in this section. That is, in the presence of semantic versioning, it is hard to specify which versions are affected by a given bug or a patch, and more generally, to understand the relation between different versions and their temporal interleaving.

To understand the issue in detail, let us consider the example of `mithril` package shown in Figure 8. This package has three major releases, all of which were concurrently maintained for some time. Because of this, the natural ordering of versions does not match with the publishing order. For example, version 1.1.7 was published on 23 Sep 2019 after version 2.0.4 which is published on 18 Aug 2019 and marked as **latest** on npm. These cases of parallel versioning make it very difficult to specify the constraint to indicate the vulnerable versions. To specify the vulnerable versions for `mithril`, Snyk database uses this complex constraint: $>= 1.0.0 < 1.1.7, >= 2.0.0 < 2.0.3$. But as version 1.1.7 is released later than 2.0.3, it is not trivial for a developer to determine whether the version is vulnerable or not. We identify multiple release candidate versions, e.g., 2.0.0-rc.0 that are actually vulnerable, but that are not flagged accordingly by the Snyk website, for this vulnerability. That is because they are not captured in the constrained above. Nonetheless, for `mithril`, there is an additional security advisory, corresponding to a different attack vector, that marks all the identified inconsistent versions as vulnerable.

Implications The findings in this section show that specifying and maintaining constraints depicting the versions affected by a given vulnerability is a very difficult problem, for which further automation is needed.

4.3 Flawed Fixes

We observe that there are variations of the existing attack vectors used in our exploits, hence we suspect that some of the fixes for the vulnerabilities we considered may be overfitted to their corresponding proof-of-concept provided in the original advisory. For example, for a prototype pollution vulnerability, one can pollute the global object using both the paths `obj.__proto__` and `obj.constructor.prototype`. Hence, the fix must account for both these two attack vectors.

Setup To study this hypothesis, we design three simple mutations, shown in the first column of Table 3. The last two capture the case described earlier, while the first mutation corresponds to a type confusion problem[3]. We then update

[3]https://snyk.io/blog/remediate-javascript-type-confusion-bypassed-input-validation/

Table 3. Mutations for identifying problematic fixes, and the zero-day vulnerabilities identified by each mutation.

Mutation	Security advisories
`"__proto__"` → `["__proto__"]`	CVE-2021-23518, CVE-2021-23760, CVE-2021-23507 CVE-2021-23497, CVE-2021-23460, CVE-2021-23558 CVE-2022-25354, CVE-2022-25296, CVE-2022-25352
`"__proto__"` → `"constructor.prototype"`	CVE-2022-22143, CVE-2022-24279
`"__proto__": {...}` → `"constructor": {"prototype": {...}}`	CVE-2021-23470

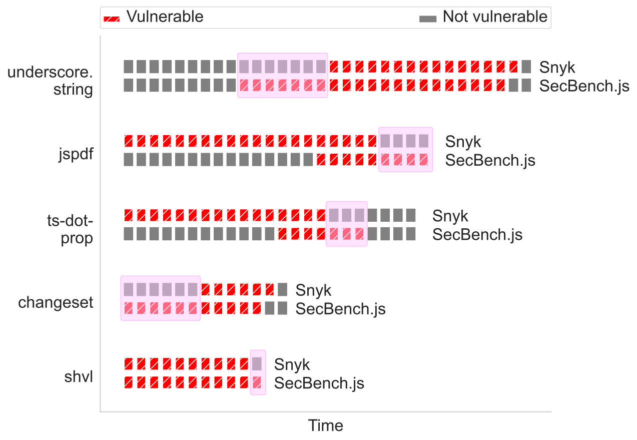

Figure 7. Example disagreement between our dynamic results and the Snyk database. Each box represents a version of the package. For each package, the lower line shows the assessment in SecBench.js, while the upper line shows the available vulnerable version information in the Snyk database. The two lines are syncronized in the sense that two boxes corresponding to the same x-point depict the same package version. The pink overlay points to the mismatch between the SecBench.js and the Snyk database.

all the vulnerable packages in SecBench.js to their latest versions and only consider those packages for which the original exploit does not work, i.e., they are supposedly fixed. We then apply each of the three mutations above to the original exploit and rerun the modified exploit. If the test succeeded, we identified a flawed fix for the corresponding package. We only consider prototype pollution exploits for this experiment.

Results In total, we find thirteen, four, and one zero-day vulnerabilities for the three mutations, respectively. We reported all these issues to the maintainers and, until the time of writing, we got assigned twelve CVEs assigned for our findings, as shown in Table 3. For two cases, there was a concurrent disclosure pending for the same issue, and the rest are still in the disclosure process.

Let us consider the case of `convict`, a popular package from Mozilla for managing configuration files. In response to the original vulnerability report for this package, the authors deployed a fix[4] in the `set` method, by including the `if` statement below:

```
const path = k.split('.')
const childKey = path.pop()
const pKey = path.join('.')
if (!(pKey == '__proto__' || pKey == '
    constructor' || pKey == 'prototype')) {
 const parent = walk(this._instance, pKey, true)
 parent[childKey] = v
}
```

The check prevents writing paths like `"__proto__.x"`, but not composed ones like `"constructor.prototype.x"`, where `pKey` is assigned `"constructor.prototype"`. This value does not match any of the checked literals inside the `if` condition. In response to our report, the maintainers deployed a more sophisticated fix that also considers this additional attack vector. Nonetheless, while writing this manuscript, in an attempt to provide information about the deployed patch, we noticed that the fix is still bypassable. This shows that developing a perfect fix for prototype pollution vulnerabilities is far from trivial. We will report this additional finding to the maintainers.

Implications An alternative to the experiments described in this section would be to manually analyze all the deployed patches for the 192 prototype pollutions vulnerabilities in our suite. While this is doable, the effort would be considerably higher than writing the simple mutations in Table 3 and running the suite three times. Moreover, we can run such experiments on a regular basis, for all the future versions of the target packages, to detect possible regressions. Thus, we conclude that an executable vulnerability database like SecBench.js is very useful for monitoring regressions and deployed fixes.

5 Discussion

This section discusses the application of SecBench.js in practice, its relationship with other language ecosystems, and some of its key limitations.

[4] https://github.com/mozilla/node-convict/commit/688c46afe099b445126 65dee6263eacd9f4f71a8

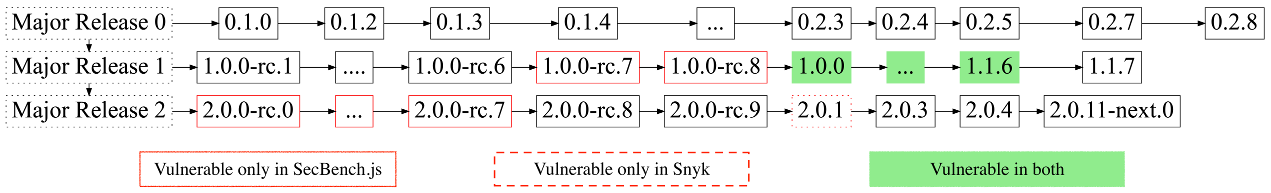

Figure 8. Example disagreement for `mithril` package in between our dynamic results and the Snyk database.

Applying SecBench.js We see several opportunities for applying SecBench.js in today's security research. First, SecBench.js is well-suited for evaluating mitigation techniques: a plethora of security systems [13, 45, 50, 51] aims to mitigate or eliminate unintended behaviors during the execution of a program. SecBench.js can be used to empirically characterize the success of these systems—including both mechanisms and policies safeguarding program execution. SecBench.js's ability to trigger vulnerabilities and confirm the anticipated side-effects via runtime oracles is critical here, including the decision to offer automation and minimize other side-effects that could interfere with mitigation techniques.

Second, SecBench.js can be used to evaluate both static and dynamic vulnerability detection techniques—*i.e.*, whether a tool or a system designed to infer legitimate behavior or detect certain classes of attacks indeed succeeds in such inference or detection. The existence of both vulnerable and non-vulnerable versions of the code in SecBench.js is important for this goal, as they can be used to characterize precision and recall—*i.e.*, metrics related to both false positives and false negatives.

Finally, SecBench.js can also be used for in-depth studies and analyses of real-world vulnerabilities and the features that enable such vulnerabilities in practice—*e.g.*, code and object complexity or source-target distance in the object graph. SecBench.js's dynamic study and fixed versions across the entire vulnerability set can form the basis of a corpus for automated such studies and analyses.

Relation to other languages and ecosystems The choice of a particular language and runtime environment—server-side JavaScript and Node.js—necessarily affects the classes of threats part of a benchmark suite. Server-side JavaScript means that a vulnerable component does not run in the sandboxed environment of a web browser. A vulnerable component, once exploited, can thus have serious effects in the broader environment in which the program is executing—including reading environment variables, writing files, spawning new processes, and setting up network connections. Many of these threat classes are common in other languages and ecosystems, such as Python (PyPI), Ruby (Rubygems), and Java (Maven Central).

A few threats part of SecBench.js are not commonly found in other environments and are due to design decisions related to the semantics and implementation of JavaScript. For example, prototype pollution attacks are due to the combination of mutable intrinsics and runtime resolution available in the JavaScript language. As another example, ReDoS attacks are due to cooperative task scheduling present in the JavaScript runtime environment. These behaviors exist in other environments (mutable intrinsics in Smalltalk; cooperative scheduling in Lua) but have not received the attention they have received in JavaScript. Less similar attacks are possible through alternative means corresponding to the mechanisms available in other languages—*e.g.*, runtime reflection and metaprogramming in languages such as Java, Ruby, and Python.

At the same time, some other classes of vulnerabilities that are possible or even common in other environments are not part of SecBench.js. One example is vulnerabilities stemming from the lack of memory safety, common in components developed in memory-unsafe languages such as C and C++. Another example is cross-site scripting attacks, common in components targeting front-end web applications.

Limitations Our suite is limited by the diversity of its components, the classes of threats it includes, and potential interference with mitigation mechanisms.

One set of limitations is related to the diversity of available components: a component can be exploited in a variety of different ways and thus, multiple SecBench.js exploits target the same or similar packages. We view such diversity limitations as affecting primarily the evaluation of performance metrics rather than those of security—as an attacker would need only one of the possible exploits to succeed in order to take control of a system. Therefore, we anticipate the users of SecBench.js to supplant the security evaluation of their systems with performance results gathered using additional benchmarks or applications appropriate for their target domains.

Another limitation is related to the classes of threats included in SecBench.js, constrained by the target language, environment, and requirements (§1). This limitation is to some extent unavoidable, as any real-world collection of threats would necessarily be finite and informed by existing, real-world constraints. Thus we view SecBench.js as a starting set—one informed by the evaluation of our own systems—open to contributions from the broader community and leveraging the structure and automation supporting SecBench.js today, usable by any new vulnerabilities extending the current set.

SecBench.js is intended for research and thus provides infrastructure to automate, instrument, and validate the execution of the benchmarks included in the set. There is a small chance that this infrastructure might accidentally interfere with security policies or mechanisms associated with the artifact being evaluated. For example, SecBench.js's creation of new processes might interfere with systems that detect or mitigate process-related limits; and its execution in a dedicated container environment might interfere with policies related to limitation and prioritization of operating-system resources. Some of these limitations would be present in any evaluation infrastructure—*i.e.*, even with ad hoc benchmarks—while others can be ameliorated through careful engineering of the artifact under evaluation.

6 Related work

Vulnerability datasets There are different kinds of vulnerability datasets. One of them is benchmarks of vulnerable programs aimed to be used for evaluating static analyzers or fuzzers [7, 17, 25]. The most closely related such benchmark is Magma [25], which is also built from real-world vulnerabilities and comes with inputs to exploit them, but targets C instead of JavaScript. Moreover, their exploitation is limited to a crash, while our testing oracle assert the success of as security-relevant action. The AEG exploit generation system [2] aims at finding vulnerabilities and generating exploits automatically. Another kind of dataset is large-scale datasets extracted in an automated manner, e.g., from version histories [19, 20, 22, 56], intended as training data for machine learning-based vulnerability detection. Due to their automated creation, these datasets do not come with exploits and suffer from some degree of noise (e.g., 53% true positives based on manual inspection [56]). Finally, a third kind of dataset offers manually validated vulnerabilities and exploits for them [34, 39], similar to SecBench.js, but none of them targets JavaScript. Beyond vulnerabilities, other benchmark suites [4, 5, 26] are mainly designed to study a specific program area outside software security. To the best our knowledge, we are the first to construct a benchmark of executable, real-world vulnerabilities in JavaScript code.

Bug benchmarks Looking beyond the security domain, there are various benchmarks of general bugs, such as Defects4J [27], Bugs.jar [42], BugSwarm [49] for Java, BugsJS for JavaScript [24], and a set of JavaScript performance bugs [43]. Other benchmarks focus on concurrency bugs [32, 55], high-impact bugs [37], and non-functional bugs [41]. While many of these benchmarks also provide inputs to trigger the bugs, they do not focus on vulnerabilities.

npm and other package ecosystems The prevalence of vulnerabilities in npm and other package ecosystems has motivated various studies and techniques to understand and identify ecosystem-level security issues. Zimmermann et al. [57] study ecosystem-level security threats in npm. Others study the impact of vulnerabilities on package dependency network [15], the phenomenon of "trivial" packages in npm [1], or the impact of ReDoS vulnerabilities [12]. Supply chain attacks are another problem of specific interest [18, 52]. Pashchenko et al. [38] report on an interview-based study to understand the (lack of) dependency management. All the above highlights the importance of security threats in large-scale package ecosystems. SecBench.js will help address these threats by providing a benchmark for evaluating future detection and mitigation tools.

JavaScript security There are various techniques for detecting JavaScript vulnerabilities and for mitigating their exploitation, of which we discuss a representative sample. Many techniques detect a particular kind of vulnerability, such as injection vulnerabilities [21, 45], hidden property attacks [54], ReDoS vulnerabilities [44], and prototype polution [30]. Others provide more general detection techniques, e.g., in the form of static extraction of taint specifications [46], dynamic taint analysis [28], and graph-based vulnerability detection [31]. There are mitigations against ReDoS [11, 13, 14], in the form of compartementalization [50], privilege reduction [51], and debloating of packages [29]. Beyond the JavaScript code itself, security-relevant bugs in the language implementation are another concern [8, 16]. We envision SecBench.js to help compare and improve techniques that detect JavaScript vulnerabilities and that mitigate their exploitation.

7 Conclusion

Computer science research and development depend crucially on benchmarks—a common foundation for evaluating techniques, systems, and solutions. Security research for server-side JavaScript currently lacks a comprehensive set of real-world executable benchmarks, collected and analyzed systematically. As a result, researchers are forced to evaluate their contributions ad hoc, hampering the direct comparison among different techniques—a problem we have repeatedly faced ourselves and heard from others when developing systems targeting defensive software security. This paper makes a first step towards addressing this problem by introducing SecBench.js—a benchmark suite of vulnerabilities and executable exploits for server-side JavaScript. SecBench.js contains 600 real-world vulnerabilities and executable exploits, spread across several threat classes. SecBench.js is offered in a fully automated package that contains (1) test oracles for automatically verifying the success of each exploit, (2) pointers to the fixed versions of each benchmark, and (3) additional metadata, such as contextual information—all enabled by a manual study of each vulnerability in the benchmark set. We perform several experiments to show the usefulness of the suite: identify flawed fixes and inaccuracies in the security advisories, extract vulnerability location, and identify deployed patches. As a result of these experiments, we uncover 20 zero-day vulnerabilities, for which we were awarded 12 CVEs. We believe that these initial results show the potential of executable vulnerability databases like SecBench.js, and we hope that the community will join in the effort of extending and maintaining this benchmark suite.

References

[1] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. 2017. Why Do Developers Use Trivial Packages? An Empirical Case Study on npm. In *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*.

[2] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. 2011. AEG: Automatic Exploit Generation. In *Network and Distributed System Security Symposium (NDSS)*.

[3] Zhihao Bai, Ke Wang, Hang Zhu, Yinzhi Cao, and Xin Jin. 2021. Runtime Recovery of Web Applications under Zero-Day ReDoS Attacks. In *Symposium on Security and Privacy (S&P)*.

[4] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: characterization and architectural implications. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

[5] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Bok Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo benchmarks: java benchmarking development and analysis. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

[6] Marcel Böhme and Abhik Roychoudhury. 2014. CoREBench: studying complexity of regression errors. In *International Symposium on Software Testing and Analysis (ISSTA)*.

[7] Tim Boland and Paul E. Black. 2012. Juliet 1.1 C/C++ and Java Test Suite. *Computer* (2012).

[8] Fraser Brown, Shravan Narayan, Riad S. Wahby, Dawson R. Engler, Ranjit Jhala, and Deian Stefan. 2017. Finding and Preventing Bugs in JavaScript Bindings. In *Symposium on Security and Privacy (S&P)*.

[9] Brian Caswell. [n. d.]. Cyber Grand Challenge Corpus. http://www.lungetech.com/cgc-corpus/

[10] Gregory Cohen, Saeed Afshar, Jonathan Tapson, and André van Schaik. 2017. EMNIST: Extending MNIST to handwritten letters. In *International Joint Conference on Neural Networks (IJCNN)*.

[11] James C. Davis. 2019. Rethinking Regex engines to address ReDoS. In *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*.

[12] James C. Davis, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. 2018. 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)*.

[13] James C Davis, Francisco Servant, and Dongyoon Lee. 2021. Using Selective Memoization to Defeat Regular Expression Denial of Service (ReDoS). In *Symposium on Security and Privacy (S&P)*.

[14] James C. Davis, Eric R. Williamson, and Dongyoon Lee. 2018. A Sense of Time for JavaScript and Node.js: First-Class Timeouts as a Cure for Event Handler Poisoning. In *USENIX Security Symposium*.

[15] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the impact of security vulnerabilities in the npm package dependency network. In *International Conference on Mining Software Repositories (MSR)*.

[16] Sung Ta Dinh, Haehyun Cho, Kyle Martin, Adam Oest, Kyle Zeng, Alexandros Kapravelos, Gail-Joon Ahn, Tiffany Bao, Ruoyu Wang, Adam Doupé, et al. 2021. Favocado: Fuzzing the Binding Code of JavaScript Engines Using Semantically Correct Test Cases. In *Network and Distributed System Security Symposium (NDSS)*.

[17] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, William K. Robertson, Frederick Ulrich, and Ryan Whelan. 2016. LAVA: Large-Scale Automated Vulnerability Addition. In *IEEE Symposium on Security and Privacy (S&P)*.

[18] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. 2021. Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages. In *Network and Distributed System Security Symposium (NDSS)*.

[19] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. In *International Conference on Mining Software Repositories (MSR)*.

[20] Rudolf Ferenc, Péter Hegedüs, Péter Gyimesi, Gábor Antal, Dénes Bán, and Tibor Gyimóthy. 2019. Challenging machine learning algorithms in predicting vulnerable JavaScript functions. In *Proceedings of the 7th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering, RAISE@ICSE 2019, Montreal, QC, Canada, May 28, 2019*, Tim Menzies and Burak Turhan (Eds.). IEEE / ACM, 8–14. https://doi.org/10.1109/RAISE.2019.00010

[21] François Gauthier, Behnaz Hassanshahi, and Alexander Jordan. 2018. AFFOGATO: runtime detection of injection attacks for Node.js. In *International Symposium on Software Testing and Analysis (ISSTA)*.

[22] Antonios Gkortzis, Dimitris Mitropoulos, and Diomidis Spinellis. 2018. VulinOSS: a dataset of security vulnerabilities in open-source systems. In *International Conference on Mining Software Repositories (MSR)*.

[23] Liang Gong. 2018. *Dynamic Analysis for JavaScript Code.* Ph.D. Dissertation. University of California, Berkeley.

[24] Péter Gyimesi, Béla Vancsics, Andrea Stocco, Davood Mazinanian, Árpád Beszédes, Rudolf Ferenc, and Ali Mesbah. 2019. BugsJS: a Benchmark of JavaScript Bugs. In *Conference on Software Testing, Validation and Verification, (ICST)*.

[25] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2021. Magma: A Ground-Truth Fuzzing Benchmark. In *International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, Longbo Huang, Anshul Gandhi, Negar Kiyavash, and Jia Wang (Eds.).

[26] John L. Henning. 2006. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News* (2006).

[27] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *International Symposium on Software Testing and Analysis (ISSTA)*.

[28] Rezwana Karim, Frank Tip, Alena Sochurkova, and Koushik Sen. 2018. Platform-independent dynamic taint analysis for JavaScript. *IEEE Transactions on Software Engineering* (2018).

[29] Igibek Koishybayev and Alexandros Kapravelos. 2020. Mininode: Reducing the Attack Surface of Node. js Applications. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*.

[30] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. 2021. Detecting Node.js prototype pollution vulnerabilities via object lookup analysis. In *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*.

[31] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. 2022. Mining Node.js Vulnerabilities via Object Dependence Graph and Query. In *USENIX Security Symposium*.

[32] Ziyi Lin, Darko Marinov, Hao Zhong, Yuting Chen, and Jianjun Zhao. 2015. JaConTeBe: A Benchmark Suite of Real-World Java Concurrency Bugs. In *International Conference on Automated Software Engineering*

(ASE).

[33] Yinxi Liu, Mingxue Zhang, and Wei Meng. 2021. Revealer: Detecting and Exploiting Regular Expression Denial-of-Service Vulnerabilities. In *Symposium on Security and Privacy (S&P)*.

[34] Joydeep Mitra and Venkatesh-Prasad Ranganath. 2017. Ghera: A Repository of Android App Vulnerability Benchmarks. In *International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE)*.

[35] Benjamin Barslev Nielsen, Behnaz Hassanshahi, and François Gauthier. 2019. Nodest: feedback-driven static analysis of Node.js applications. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, (FSE)*.

[36] Benjamin Barslev Nielsen, Martin Toldam Torp, and Anders Møller. 2021. Modular call graph construction for security scanning of Node.js applications. In *International Symposium on Software Testing and Analysis (ISSTA)*.

[37] Masao Ohira, Yutaro Kashiwa, Yosuke Yamatani, Hayato Yoshiyuki, Yoshiya Maeda, Nachai Limsettho, Keisuke Fujino, Hideaki Hata, Akinori Ihara, and Ken-ichi Matsumoto. 2015. A Dataset of High Impact Bugs: Manually-Classified Issue Reports. In *International Conference on Mining Software Repositories (MSR)*.

[38] Ivan Pashchenko, Duc Ly Vu, and Fabio Massacci. 2020. A Qualitative Study of Dependency Management and Its Security Implications. In *Conference on Computer and Communications Security (CCS)*.

[39] Serena Elisa Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and Cédric Dangremont. 2019. A manually-curated dataset of fixes to vulnerabilities of open-source software. In *International Conference on Mining Software Repositories (MSR)*.

[40] Meikel Pöss and Chris Floyd. 2000. New TPC Benchmarks for Decision Support and Web Commerce. *SIGMOD Rec.* (2000).

[41] Aida Radu and Sarah Nadi. 2019. A dataset of non-functional bugs. In *International Conference on Mining Software Repositories (MSR)*.

[42] Ripon K. Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul R. Prasad. 2018. Bugs.jar: a large-scale, diverse dataset of real-world Java bugs. In *International Conference on Mining Software Repositories (MSR)*, Andy Zaidman, Yasutaka Kamei, and Emily Hill (Eds.).

[43] Marija Selakovic and Michael Pradel. 2016. Performance Issues and Optimizations in JavaScript: An Empirical Study. In *International Conference on Software Engineering (ICSE)*. 61–72.

[44] Cristian-Alexandru Staicu and Michael Pradel. 2018. Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers. In *USENIX Security Symposium*.

[45] Cristian-Alexandru Staicu, Michael Pradel, and Benjamin Livshits. 2018. SYNODE: Understanding and Automatically Preventing Injection Attacks on NODE.JS. In *Network and Distributed System Security Symposium (NDSS)*.

[46] Cristian-Alexandru Staicu, Martin Toldam Torp, Max Schäfer, Anders Møller, and Michael Pradel. 2020. Extracting taint specifications for JavaScript libraries. In *International Conference on Software Engineering (ICSE)*.

[47] Johannes Stallkamp, Marc Schlipsing, Jan Salmen, and Christian Igel. 2011. The German Traffic Sign Recognition Benchmark: A multi-class classification competition. In *International Joint Conference on Neural Networks (IJCNN)*.

[48] Matthew Taylor, Ruturaj K. Vaidya, Drew Davidson, Lorenzo De Carli, and Vaibhav Rastogi. 2020. Defending Against Package Typosquatting. In *Network and Distributed System Security Symposium (NDSS)*.

[49] David A. Tomassi, Naji Dmeiri, Yichen Wang, Antara Bhowmick, Yen-Chuan Liu, Premkumar T. Devanbu, Bogdan Vasilescu, and Cindy Rubio-González. 2019. BugSwarm: mining and continuously growing a dataset of reproducible failures and fixes. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 339–349. https://doi.org/10.1109/ICSE.2019.00048

[50] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. 2018. BreakApp: Automated, Flexible Application Compartmentalization. In *Network and Distributed System Security Symposium, (NDSS)*.

[51] Nikos Vasilakis, Cristian-Alexandru Staicu, Grigoris Ntousakis, Konstantinos Kallas, Ben Karel, André DeHon, and Michael Pradel. 2021. Preventing Dynamic Library Compromise on Node.js via RWX-Based Privilege Reduction. In *Conference on Computer and Communications Security (CCS)*.

[52] Duc Ly Vu, Ivan Pashchenko, Fabio Massacci, Henrik Plate, and Antonino Sabetta. 2020. Towards Using Source Code Repositories to Identify Software Supply Chain Attacks. In *Conference on Computer and Communications Security (CCS)*.

[53] Feng Xiao, Jianwei Huang, Yichang Xiong, Guangliang Yang, Hong Hu, Guofei Gu, and Wenke Lee. 2021. Abusing Hidden Properties to Attack the Node.js Ecosystem. In *USENIX Security Symposium*.

[54] Feng Xiao, Jianwei Huang, Yichang Xiong, Guangliang Yang, Hong Hu, Guofei Gu, and Wenke Lee. 2021. Abusing Hidden Properties to Attack the Node.js Ecosystem. In *USENIX Security Symposium*.

[55] Ting Yuan, Guangwei Li, Jie Lu, Chen Liu, Lian Li, and Jingling Xue. 2021. GoBench: A Benchmark Suite of Real-World Go Concurrency Bugs. In *International Symposium on Code Generation and Optimization, (CGO)*.

[56] Yunhui Zheng, Saurabh Pujar, Burn L. Lewis, Luca Buratti, Edward A. Epstein, Bo Yang, Jim Laredo, Alessandro Morari, and Zhong Su. 2021. D2A: A Dataset Built for AI-Based Vulnerability Detection Methods Using Differential Analysis. In *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2021, Madrid, Spain, May 25-28, 2021*. IEEE, 111–120. https://doi.org/10.1109/ICSE-SEIP52600.2021.00020

[57] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small World with High Risks: A Study of Security Threats in the npm Ecosystem. In *USENIX Security Symposium*.