Work-in-progress: JAVULIN: Scalable Vulnerability Injection for JavaScript Web Applications

Dominic Troppmann
CISPA Helmholtz Center
for Information Security
dominic.troppmann@cispa.de

Cristian-Alexandru Staicu
Endor Labs
cris.staicu@gmail.com

Aurore Fass
Inria Centre at
Université Côte d'Azur
aurore.fass@inria.fr

***Abstract*—The detection of security vulnerabilities in JavaScript web applications is a rapidly advancing field, with new vulnerability-detection techniques emerging each year. However, the lack of realistic, large-scale ground-truth datasets of vulnerabilities remains a significant barrier that hinders the scientific progress in this area. Existing benchmarks are either limited to code fragments or libraries, rely on synthetic bugs that may not fully reflect the complexity of real-world faults, or require labor-intensive manual curation, severely hampering reproducibility and scalability. As a result, the true capabilities and limitations of modern vulnerability detection tools for JavaScript remain largely unknown. In this work, we present JAVULIN, the first semi-automated framework for generating datasets of application-level JavaScript vulnerabilities at scale, using vulnerability injection. By transplanting known library-level flaws into diverse, dynamically analyzed target web applications, our approach creates proof-carrying, realistic benchmarks that capture the genuine complexity of real-world JavaScript vulnerabilities. We systematically instrument and explore each target application to identify and select viable injection points, automatically compile a portfolio containing hundreds of seed vulnerabilities, and fully automate the injection of vulnerabilities and the generation of proof-of-concept HTTP-level exploits. With this work, we aim to enable the creation of large, realistic JavaScript security benchmarks that strive to challenge state-of-the-art analysis at an unprecedented scale.**

I. INTRODUCTION

In today's software-driven society, web applications provide essential services in banking, healthcare, and communication, exposing billions of users to severe consequences if systems are compromised. In an effort to make these systems more secure, researchers put significant effort into developing new approaches to vulnerability detection and analysis, resulting in a constant stream of vulnerability scanners for JavaScript in recent years [1], [2], [3], [4], [5], [6], [7], [8], [9]. Yet, evaluating, benchmarking, and comparing such tools remains notoriously difficult, primarily due to the lack of a universal large-scale, realistic *ground-truth* corpora of vulnerable web applications.

Historically, the research community has focused considerable effort on synthetic benchmark suites in low-level languages, such as C [10], [11], [12], [13], where artificial bugs are automatically injected into open-source programs. Such approaches enable the generation of large corpora at a relatively low cost, compared to manually curating a high-quality dataset of real-world faults. While the datasets generated by these prior approaches have spurred significant advances in vulnerability detection, their applicability is limited in both scope and realism. In particular, being developed for and limited to C, they do not address the dynamic nature and unique security challenges of JavaScript, the de facto language of the modern Web. Bringing similar rigor to JavaScript vulnerability analysis is essential, as the language's weak typing, prototype-based inheritance, and pervasive use of third-party packages present a broad and subtle attack surface [14], [15], [16], [6]. Yet, the JavaScript community remains underserved, with the few existing datasets [17], [18] being limited in scalability due to their reliance on manual curation and real-world bugs and vulnerabilities, which are scarce and thus difficult to find in the first place.

To motivate the potential benefits of generating datasets via vulnerability injection, we want to highlight SCA Goat[1]. SCA Goat is a deliberately vulnerable Java application that utilizes vulnerable JAR dependencies in its development code. Its main purpose is to give users a hands-on learning opportunity and a chance to experiment with detecting the included vulnerabilities. SCA Goat covers ten "critical" and "high-severity" CVEs. Despite the small scale of the dataset, considerable manual effort was required to curate it, with the repository featuring more than 100 commits over nearly 19 months. This example illustrates the limitations of manual curation and motivates us to overcome them by leveraging (semi-)automated vulnerability injection.

In this work, we present JAVULIN, the first semi-automated, scalable framework for generating large corpora of vulnerable JavaScript web applications using vulnerability injection. We precisely locate viable injection points by dynamically analyzing the target web applications. Then, unlike prior approaches, we inject known library-level vulnerabilities into target applications, rather than introducing artificial bugs. This produces security benchmarks with vulnerabilities that are exploitable, deeply embedded, and tailored to reflect the true complexity

Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb) 2026
27 February 2026, San Diego, CA, USA
ISBN 978-1-970672-06-0
https://dx.doi.org/10.14722/madweb.2026.23011
www.ndss-symposium.org

[1] https://github.com/harekrishnarai/Damn-vulnerable-sca

of web application logic. Crucially, our approach overcomes several key obstacles: (1) it enables reproducible vulnerabilities by automatically generating proofs of vulnerabilities for each successful injection; (2) it produces deep, non-trivial faults; (3) it matches the complexity and diversity of organic vulnerabilities by drawing from real vulnerability disclosures; and (4) it is more scalable than prior approaches relying on the (manual) curation of scarce known faults. Altogether, this enables the systematic testing, evaluation, and comparison of modern security tools in scenarios closely matching those security analysts encounter in practice.

II. Background

In this section, we review the essential concepts for evaluating vulnerability detection tools. We examine the role and construction of ground-truth datasets, highlight the specific challenges posed by JavaScript, and clarify what makes a realistic and valuable vulnerability benchmark.

A. Security Benchmarks

The evaluation of security tools hinges on the availability of *ground-truth datasets* — collections of applications with vulnerabilities whose locations and exploitability are precisely known. This allows researchers to judge the performance of detectors not only by the number of correctly identified faults (true positives), but also by the number of faults that tools miss (false negatives). Several core criteria have emerged to build useful evaluation corpora. The following properties are essential for a high-quality security benchmark [10], [12]:

- **Realism:** Bugs should mimic the subtlety and exploitability of those found in deployed software.
- **Diversity:** Benchmarks should span multiple bug classes, codebases, and application domains.
- **Scalability:** Injected faults should be cheap and plentiful to allow for large-scale corpora.
- **Reproducibility:** Each injected vulnerability must come with a *Proof of Vulnerability (PoV)*, i.e., a trigger that demonstrates its existence and exploitability.

While ground-truth datasets are crucial for reproducible, comprehensive, and comparable evaluation of vulnerability detectors, few exist in practice due to how difficult and expensive it is to curate them (manually).

B. JavaScript Challenges

JavaScript, powering the majority of web applications, presents extraordinary challenges for both detection and injection. Its highly dynamic runtime, absence of static typing, and prototype-oriented object model introduce entire classes of vulnerabilities—notably, prototype pollution [19], [16], [20]—that are structurally foreign to languages such as C or Java. Moreover, modern JavaScript applications rely extensively on asynchronous patterns and event-driven architectures, compounding the complexity of static and dynamic analysis. A further complication is JavaScript's vast ecosystem of third-party packages. These dependencies introduce indirect vulnerabilities and attack vectors, both through direct code reuse and via transitive dependency chains [15]. Together, these characteristics render prior approaches to vulnerability injection — and their resulting datasets — ill-suited for the needs of JavaScript security. Effective evaluation requires methods that consider the language's dynamism and ecosystem scale.

III. Related Work

This section highlights closely related work, focusing on available security scanners and benchmarks for JavaScript, as well as prior approaches to vulnerability injection in other programming languages.

Vulnerability Detection. In recent years, there was a constant stream of newly developed vulnerability detection tools [1], [2], [3], [4], [5], [6], [7], [8], [9]. First, a series of taint-analysis-based approaches targeting injection vulnerabilities in Node.js applications [1], [2], [3], culminating in Nodest by Nielsen et al. [4]. Nodest leverages feedback-driven taint-analysis and scales to large Node.js applications with many dependencies, something prior approaches struggle with. While the authors evaluate Nodest on the benchmarks used in these prior studies, they also include two additional Express-based applications to demonstrate the scalability of their approach. MERLIN [9] is a multi-language vulnerability detector targeting multiple vulnerability classes. While the detector supports JavaScript, the authors do not evaluate their tool on any JavaScript applications. A different line of work by Li et al. [6], [21] produced multiple static analysis tools targeting JavaScript prototype pollution (PP) vulnerabilities [20]. Although both tools are developed by the same authors and target the same vulnerability type, they are evaluated on differing datasets, i.e., a subset of NPM packages at differing dates, as well as different sets of known Node.js vulnerabilities. The datasets are also disjoint from the benchmarks used in other prior work. In summary, we note a constant stream of new tools that are evaluated on mostly disjoint datasets, making it hard to compare their performance. Furthermore, we note a distinct lack of ground-truth in most evaluations, i.e., authors only rely on the number of newly identified faults to judge their tool's performance.

JS Security Benchmarks. The second category of related work concerns efforts to curate security benchmarks for JavaScript [17], [18], [22]. BugsJS [17] comprises 453 known and manually validated JavaScript bugs in ten popular server-side Node.js programs. The bugs are manually collected from bug-fixing commits on GitHub and thoroughly inspected before being included in the benchmark, along with tests and documentation about the bug and its resolution. SecBench.js [18] is a JavaScript security benchmark containing 600 vulnerabilities, covering the five most common vulnerability classes for server-side JavaScript: path traversal, prototype pollution, command injection, denial-of-service, and code injection. The vulnerabilities are manually collected from security advisories (e.g., Snyk) and come with a proof of vulnerability demonstrating their exploitability. Finally, Brito et

al. present VulcaN [22], an execution and analysis framework which they used to manually curate a dataset containing 957 vulnerabilities in npm packages, sourced from npm advisories. While these datasets are realistic and diverse, their scalability is limited by the significant manual effort required for curation.

Synthetic Benchmarks. Next are prior approaches that generate synthetic benchmarks, i.e., datasets containing vulnerabilities that were not discovered "as-is" in the real world: Magma [12] is a ground-truth fuzzing benchmark consisting of seven target programs with 118 known bugs sourced from older versions. The approach is entirely manual, including the collection and inspection of bug reports for the target applications and *forward-porting* the selected bugs to the targets' latest versions. While this approach yields a high-quality corpus that is both realistic and diverse, the authors argue that its complexity prevents automation, as it would result in an "incomplete and error-prone technique."

LAVA [10] automatically adds vulnerabilities to target C programs by identifying unused input bytes and adding code that triggers an out-of-bounds memory access if these bytes match a specific "magic" value. The bugs produced by LAVA are cheap and plentiful. However, it is questionable how realistic the injected bugs are due to artificial dataflow introduced by the approach and the magic values used to guard the injected bugs [11], [23], [13], [24]. Furthermore, LAVA only considers one class of vulnerability, which severely limits the dataset's *diversity*. While some of these issues were addressed in a follow-up study by Sridar [11], more recent studies suggest that the LAVA-M dataset is outdated by today's standards, as recent tools can easily identify all the injected bugs [25]. EVILCODER [26] takes a slightly different approach; instead of adding vulnerable code to a target, it renders applications vulnerable by disabling security mechanisms, such as input validation and sanitizers. While these bugs are more realistic than those injected by LAVA, EVILCODER fails to provide a PoV. Hence, the exploitability of the added vulnerabilities must be manually validated, which limits the approach's scalability. Finally, Roy et al. propose APOCALYPSE [13], which relies on a combination of formal methods to inject more realistic, fair, but harder-to-find bugs into C programs, while generating PoVs.

Effectiveness of Synthetic Bugs. Several studies assess the usefulness of synthetic bugs and benchmarks [24], [23]. By running eight state-of-the-art fuzzers against 20 targets from the Rode0day [25] and LAVA-M [10] corpora for a total of 733K CPU hours, Bundt et al. showed that synthetic bugs are typically less difficult to detect than organic ones. This is mainly because most of the bugs injected are very close to the "main path", i.e., code easily covered by modern fuzzers [24]. In contrast, organic bugs tend to reside in parts of the code that are significantly more difficult to reach. This is a hard problem for any bug injection approach since adding vulnerabilities randomly to uncovered parts of the code would make it hard to generate a PoV (i.e., triggering input) for the bugs. Geng et al. similarly conclude that synthetic bugs tend to poorly reflect reality [23]. Despite these criticisms, vulnerability addition has proven useful in practice, considering that LAVA has spawned several follow-up works [11], [26], and the Lava-M dataset has been used to evaluate several vulnerability detection tools [27], [28], [29].

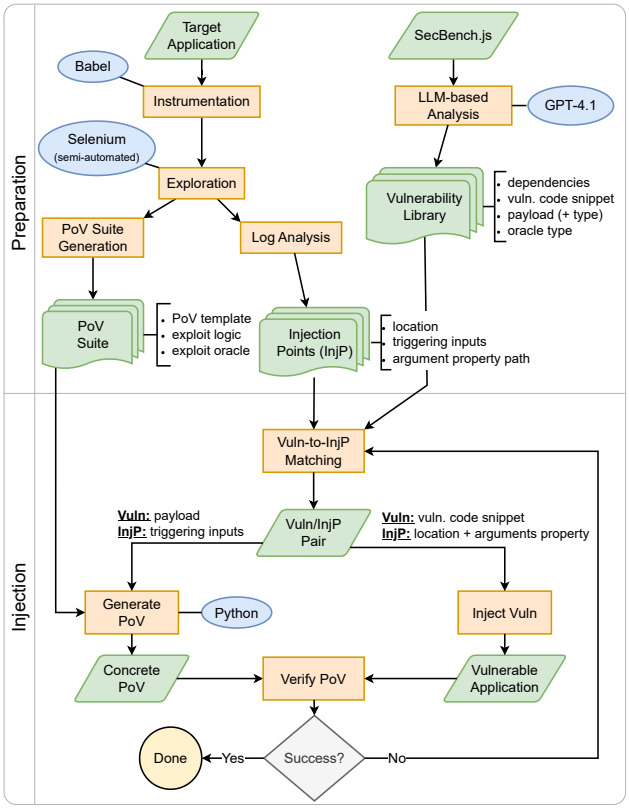

Fig. 1: Overview of our proposed approach.

IV. METHODOLOGY

In this section, we discuss our semi-automated approach for injecting known library-level JavaScript vulnerabilities into JavaScript web applications. As shown in Figure 1, our process consists of two major phases. In the first phase, we prepare all the necessary components once per application. Then, we automatically inject vulnerabilities in the second phase, generating corresponding Proofs of Vulnerability (PoV) in the process. Below, we provide a more detailed description of each phase.

A. Phase 1: Preparation

In the first phase, we set up target applications for the pipeline and compile a portfolio of seed vulnerabilities. Figure 1 shows the main components of this phase. On the left-hand side is the *dynamic analysis* of the target applications, consisting of their *instrumentation* and *exploration*. It yields information about potential injection points in the application's code, as well as means to replay certain interactions later

for generating and verifying PoVs. The second component involves compiling a portfolio of seed vulnerabilities for later injection, which we refer to as a *vulnerability library*, shown on the right-hand side.

1) Dynamic Analysis: First, we dynamically analyze the target applications to: (1) obtain a list of viable injection points, i.e., functions we know how to trigger at runtime, and (2) create a PoV suite, i.e., a collection of Python Selenium tests to replay interactions and trigger specific injection points. This part of the preparation phase is application-specific, meaning that it must be repeated once for every target application in the dataset.

Instrumentation. First, we instrument the target application using Babel [2]. Our custom Babel plugin adds a small code snippet to each function in the target application that logs the function's name, location, and `arguments` object whenever the function is triggered (see Figure 5 in the Appendix). This later allows us to identify functions reached by inputs we passed to the application during exploration, linking front-end interactions with back-end functionality and their specific code locations. The instrumentation of the source code via Babel is fully automated. However, some manual effort may be required to fine-tune Babel configurations to produce an instrumented application that builds and runs properly.

Exploration. Once we have a stable instrumented version of the target application, we manually explore it, using the Selenium IDE [3]. Selenium affords recording interactions and replaying them later via its browser automation API. We record one test case per interaction: we start the recording, navigate from the home page to, for example, a specific form, fill it out, and submit it. Then we revert the interaction if possible; for example, if we created a blog post, we delete it. Once the initial state is restored, we stop the recording. As we explore the target application, the code we injected during instrumentation records the functions triggered by our interactions, allowing us to learn which interactions with the web UI on the front-end trigger which functions on the back end. We emphasize the importance of reverting to the initial state at the end of each interaction, as any lingering modifications to the website's content or UI may interfere with subsequent interactions.

Injection Points. After running the dynamic analysis, we analyze the logs produced by the instrumentation code during the exploration to learn which functions were reached by the inputs we manually provided through our interactions with the front-end. We use regular expressions to identify function calls that contain any of the inputs passed to the application in their arguments object. We store a list of triggered functions in a JSON file, along with other useful information, such as the function name, file location, the number of times the function was triggered, and the exact path to access our input inside the function's arguments object.

[2]https://babeljs.io/docs/

[3]https://www.selenium.dev/documentation/

```
test("Arbitrary code execution in access-
    policy", () => {
  expect.assertions(2);
  const fs = require("fs");
  let a = require("access-policy");
  const path = "./access-policy";
  let statements = "`; require(`fs`).
      writeFileSync(`" + path + "`,``);//";
  data = {};
  file_exist = fs.existsSync(path);
  expect(file_exist).toBe(false);
  try {
    a.encode(statements, data);
  } catch (e) {}
  file_exist = fs.existsSync(path);
  expect(file_exist).toBe(true);
  fs.unlink(path, function (err) {});
});
```

Fig. 2: Example unit test in SecBench.js. The LLM extracts the relevant parts of the code, i.e., vulnerable code snippet, payload, and oracle (type).

```
name: access-policy_3.1.0
dependencies: {"access-policy": "3.1.0"}
payload: "`; require(`fs`).writeFileSync(`{
    ORACLE_FILE}`,``);//"
vulnerable_code: let a = require("access-policy
    "); a.encode(<PAYLOAD>, {});
payload_type: string
oracle_type: file
description: Encodes and evaluates against
    provided data
```

Fig. 3: Vulnerability summary for *access-policy_3.1.0*.

PoV Suite Generation. Finally, we create a *PoV suite* for each target application. A PoV suite consists of (1) a *PoV template* and (2) several *interaction tests*. The PoV template is a parametrized Jinja2 template [4] for a Python unittest. This template contains placeholders for (1) the name of the injected vulnerability, (2) the oracle type, (3) the triggering input passed during exploration, (4) the payload to trigger the vulnerability, and (4) the interaction test case(s) to replay the interactions that trigger the function where the vulnerability was injected. These placeholders will be filled later during the *PoV generation* phase (cf. Section IV-B2). The interaction tests are Python scripts exported directly from the Selenium IDE, corresponding to the interactions performed and recorded during exploration. Selenium allows us to export test cases directly to Python, so we only need to make slight modifications to conform to our framework and improve their stability; hence, this part of the process is semi-automated.

2) Vulnerability Library: We source our seed vulnerabilities from SecBench.js [18]. As a proof of concept, we focus on command- and code-injection vulnerabilities for now, as their exploitability is straightforward to verify and successful exploitation is less likely to compromise the target application, in contrast to, e.g., prototype pollution. JAVULIN thus uses 141

[4]https://jinja.palletsprojects.com/en/stable/

```
class GhostPoV_access_policy_3.1.0(GhostPoV, unittest.TestCase): # a)
    def __init__(self, methodName="runTest"):
        super().__init__(methodName)
        self.oracle_type = "file" # b)

    def test_exploit(self):
        # Replace exploration input with payload
        self.set_payload({
            "postsettingsmetades-input-26": # c)
            "`;␣require('fs').writeFileSync(`
                path/to/oracle`,``);//" # d)
            })
        # Trigger selected injection point
        self.postsettings() # e)

if __name__ == "__main__":
    unittest.main()
```

Fig. 4: Example of a concrete PoV script.

of SecBench.js' 600 total vulnerabilities. While SecBench.js provides an executable JS unit test for each vulnerability in the dataset, the test cases are not uniform. For example, one test case may store its payload in a variable named `"payload"`, while another stores it in a variable called `"statements"`. Additionally, it is not always clear which parts of the code belong to the unit test framework and can be safely ignored, and which parts are necessary to trigger the vulnerability. To overcome this inconsistency and collect all the relevant bits of information, while getting rid of everything unnecessary, we parse each test case using GPT-4.1 to identify and extract the following pieces of information:

- **Description**: A short description about the functionality provided by the vulnerable function. This information can be used to make a more context-aware, and thus more realistic, match between vulnerability and injection point.
- **Payload**: The input passed to the vulnerable API call that triggers the vulnerability.
- **Payload Type**: The data type of the payload, e.g. `string` or `object`.
- **Vulnerable code**: Only the part of the code containing the vulnerability. Here, the concrete payload is replaced by a placeholder.
- **Oracle type**: The type of oracle used to verify if the exploit was successful, e.g., creating a file or polluting the prototype.

We store this information, as well as the vulnerable package's name, version, and dependencies, in a text file that we refer to as *vulnerability summary*. An example of a SecBench.js unit test and its corresponding vulnerability summary are shown in Figure 2 and Figure 3, respectively. Generating an entire vulnerability library of 141 seed vulnerabilities is fully automated and takes less than three minutes.

B. Phase 2: Injection

After setting up target applications and vulnerability library, we reach the injection phase, which consists of four steps: (1) matching vulnerabilities to injection points, (2) generating the PoV, (3) injecting the vulnerable code at the selected injection point, and (4) verifying the PoV to see if the injection was successful. In the following, we discuss each component in more detail.

1) Matching Vulnerabilities to Injection Points: First, we select a random combination of injection point and vulnerability. Currently, we do not process this selection further; i.e., we do not filter out combinations unlikely to be successful. An example of such combinations is when a complex payload with special characters is passed to a form field that is subject to input sanitization/validation (e.g., email addresses or valid URLs). In the future, more sophisticated matching heuristics may be adopted to automatically discard such injection attempts, thereby increasing the success rate and reducing the computation overhead. Furthermore, this step presents an opportunity to fine-tune the quality of the injections, e.g., by accounting for processed data types, context, and provided functionality of both the injection point and the vulnerable API, resulting in injections that are more likely to be viable, more realistic, and harder to detect.

2) PoV Generation: Next, we generate the corresponding *concrete PoV*. To this end, the following pieces of information from the vulnerability/InjP pair are automatically inserted in the PoV template:

a) name and version of the vulnerable package
b) oracle type
c) input that reached the injection point during exploration
d) payload to exploit the vulnerability
e) interaction(s) to trigger the injection point

Figure 4 shows an example PoV for the `access-policy` vulnerability highlighted previously. The PoV serves two purposes: first, it allows us to determine if the injection was successful. Second, it provides ground truth to the dataset by documenting injected vulnerabilities and how they can be exploited.

3) Injection and Verification: Finally, we insert the vulnerable code snippet from the vulnerability summary into the target application at the precise code location specified in the targeted injection point. We inject the vulnerability into the instrumented version of the application to avoid mapping code locations between the instrumented and original versions. Furthermore, logs produced by the instrumentation code during PoV verification can be useful for analyzing and debugging failing injection attempts. Then, we execute the previously generated PoV script, which automatically runs the exploit corresponding to the injected vulnerability and verifies its success.

V. Preliminary Results

The development of JAVULIN is still ongoing at the time of writing, as this study is a work in progress. We implemented all the core functionality and successfully conducted the preparation phase for four diverse, high-profile (more than 15k stars on GitHub) JavaScript web applications: Ghost [5],

[5]https://github.com/TryGhost/Ghost

node-red [6], linkwarden [7], and cal.com [8]. We already started experimenting with vulnerability injection at scale on Ghost. Each injection attempt, from start to finish, takes about 70 seconds on average. Most of the time is spent rebuilding the application before and after the injection attempt to restore the initial state, and booting the application to run the automatically generated PoV script. Note that this time varies per application, depending on its size and build/boot process. Of the 826 attempts we have made and recorded so far, 26 resulted in successful injections: the framework completed the injection process without crashing and verified the exploitability of the injected vulnerability by successfully running the generated PoV. Note that 150-200 of the failed attempts were due to a catastrophic failure during a single injection midway through a larger batch of 300 injections, which caused the app to stop building correctly. This caused subsequent attempts to fail by default, i.e., some of them may have been successful under normal conditions. Hence, the success rate is currently around 5%, on average. Other factors that cause attempts to fail include, but are not limited to, input sanitization, inconsistencies between the exploration and injection phases, and limitations of Selenium.

VI. Discussion

This work aims to provide the research community with a *realistic*, *diverse*, *large-scale*, and *reproducible* security benchmark for JavaScript vulnerabilities at the application-level. We argue that our approach meets all of these requirements to a certain degree, which sets it apart from prior work in the field. Besides being the first to propose an approach to inject real-world vulnerabilities into JavaScript web applications, our approach improves on each aspect individually, compared to existing related work: 1) Our dataset is more *realistic* than LAVA [10], EVILCODER [26], or APOCALYPSE [13], as we inject known real-world vulnerabilities in the target applications, instead of artificially modifying the applications to make them vulnerable. 2) Our approach promises to produce a *diverse* dataset. It already supports multiple target applications and hundreds of seed vulnerabilities spanning two distinct vulnerability classes. We plan to support further vulnerability classes in the future, such as prototype pollution. 3) Our approach is more *scalable* than any fully automated approach such as Magma [12], BugsJS [17], or SecBench.js [18] by design. This is because such a manual approach requires a constant amount of effort **per vulnerability** added to the dataset. In contrast, our approach only incurs manual effort **once per application** supported by JavulIn (Note that regular users will not notice this effort at all). Once an application is set up and ready for the injection phase, everything is fully automated, hence adding more vulnerabilities to the dataset is extremely cheap in comparison. 4) Finally, our approach affords *reproducibility by design*, as any successful injection comes with an automatically generated proof of vulnerability. This is not only useful for verifying whether the injection was successful, but it can also provide crucial insights when evaluating security scanners using the dataset. Some prior approaches to vulnerability injection lack such a feature [26]. To the best of our knowledge, there is no prior approach to generating reproducible exploits in the context of modern, highly dynamic web applications.

[6]https://github.com/node-red/node-red

[7]https://github.com/linkwarden/linkwarden

[8]https://github.com/calcom/cal.com

VII. Limitations & Future Work

In this section, we discuss some of JavulIn's current limitations and potential directions for future work:

1) Stability of the injection process. Critical failures can happen occasionally, for instance, when the build process fails or when an injection attempt fundamentally breaks the application. Ideally, we want our system to recover from such failures; that is, it must guarantee that the application returns to its initial state before each injection attempt. Otherwise, critical failures can cause subsequent attempts to fail by default, e.g., because the app no longer boots up correctly, preventing PoV verification.

2) Stability of selenium test cases. Interferences between subsequent injection attempts are another problem. The Selenium test cases used during exploration and in the generated PoVs assume a clean "zero" state of the application. We design the tests such that they revert any interactions before terminating. This works when running each test in isolation, but may fail when attempting multiple injections in sequence, for instance, if the target application crashes while running a PoV before it reaches the part where it reverts the actions it performed. Thus, the PoV cannot restore the content/UI to its initial state, which can interfere with subsequent injections, causing them to fail by default.

3) Seed Vulnerabilities. SecBench.js [18] – our current source of seed vulnerabilities – is no longer actively maintained, which means our tool does not consider more recently discovered vulnerabilities. To overcome this limitation, we envision sourcing seed vulnerabilities directly from CVEs and vulnerability reports in the future, using the existing LLM-based infrastructure to compile vulnerability summaries (not considering necessary adjustments to the current prompts).

VIII. Conclusion

In this paper, we present JavulIn, a semi-automated framework for injecting real-world package-level vulnerabilities into JavaScript web applications. With this approach, we aim to provide the research community with large-scale, realistic JavaScript security benchmarks, including ground truth, to facilitate a more comprehensive and fair evaluation of security scanners in the future. While development is still ongoing and several challenges remain to be addressed, initial experiments have already yielded promising results, with several dozen successful injections requiring minimal further manual effort. We plan to add more components to JavulIn to give users much more fine-grained control over the generated dataset in the future.

IX. Ethics Considerations

While the main goal of our research is to provide the research community with large-scale security benchmarks for JavaScript, it produces knowledge and a tool that pose a certain risk of dual use. We acknowledge this risk and concerns about potential misuse of JaVuLIn, and plan to take necessary steps to prevent this in the future, such as granting access to the source code and dataset only to vetted parties. Furthermore, we note that our process only utilizes publicly accessible vulnerabilities and payloads that are already known and implants these vulnerabilities in locally hosted applications, which limits the potential impact on any third parties.

Acknowledgment

This work was conducted in the scope of a dissertation at the Saarbrücken Graduate School of Computer Science.

References

[1] R. Karim, F. Tip, A. Sochrková, and K. Sen, "Platform-independent dynamic taint analysis for javascript," *IEEE Transactions on Software Engineering*, vol. 46, no. 12, pp. 1364–1379, 2018.

[2] F. Gauthier, B. Hassanshahi, and A. Jordan, "Affogato: runtime detection of injection attacks for node. js," in *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, 2018, pp. 94–99.

[3] C.-A. Staicu and M. Pradel, "Synode: Understanding and automatically preventing injection attacks on node. js." 2018.

[4] B. B. Nielsen, B. Hassanshahi, and F. Gauthier, "Nodest: feedback-driven static analysis of node. js applications," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 455–465.

[5] B. Hassanshahi, H. Lee, and P. Krishnan, "Gelato: Feedback-driven and guided security analysis of client-side web applications," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 618–629.

[6] S. Li, M. Kang, J. Hou, and Y. Cao, "Detecting node. js prototype pollution vulnerabilities via object lookup analysis," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 268–279.

[7] M. Kang, Y. Xu, S. Li, R. Gjomemo, J. Hou, V. Venkatakrishnan, and Y. Cao, "Scaling javascript abstract interpretation to detect and exploit node. js taint-style vulnerability," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 1059–1076.

[8] B. Eriksson, A. Stjerna, R. De Masellis, P. Rüemmer, and A. Sabelfeld, "Black ostrich: Web application scanning with string solvers," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 549–563.

[9] A. Figueiredo, T. Lide, D. Matos, and M. Correia, "Merlin: multi-language web vulnerability detection," in *2020 IEEE 19th International Symposium on Network Computing and Applications (NCA)*. IEEE, 2020, pp. 1–9.

[10] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "Lava: Large-scale automated vulnerability addition," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 110–121.

[11] R. Sridhar, "Adding diversity and realism to lava, a vulnerability addition system," Ph.D. dissertation, Massachusetts Institute of Technology, 2018.

[12] A. Hazimeh, A. Herrera, and M. Payer, "Magma: A ground-truth fuzzing benchmark," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 4, no. 3, pp. 1–29, 2020.

[13] S. Roy, A. Pandey, B. Dolan-Gavitt, and Y. Hu, "Bug synthesis: Challenging bug-finding tools with deep faults," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 224–234.

[14] D. Troppmann, A. Fass, and C.-A. Staicu, "Typed and confused: Studying the unexpected dangers of gradual typing," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 1858–1870.

[15] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, "Small world with high risks: A study of security threats in the npm ecosystem," in *28th USENIX Security symposium (USENIX security 19)*, 2019, pp. 995–1010.

[16] M. Shcherbakov, M. Balliu, and C.-A. Staicu, "Silent spring: Prototype pollution leads to remote code execution in node. js," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 5521–5538.

[17] P. Gyimesi, B. Vancsics, A. Stocco, D. Mazinanian, A. Beszédes, R. Ferenc, and A. Mesbah, "Bugsjs: a benchmark of javascript bugs," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2019, pp. 90–101.

[18] M. H. M. Bhuiyan, A. S. Parthasarathy, N. Vasilakis, M. Pradel, and C.-A. Staicu, "Secbench. js: An executable security benchmark suite for server-side javascript," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1059–1070.

[19] H. Y. Kim, J. H. Kim, H. K. Oh, B. J. Lee, S. W. Mun, J. H. Shin, and K. Kim, "Dapp: automatic detection and analysis of prototype pollution vulnerability in node. js modules," *International Journal of Information Security*, vol. 21, no. 1, pp. 1–23, 2022.

[20] O. Arteau, "Prototype pollution attack in nodejs application," in *NorthSec*. Olivier Arteau, 2018.

[21] S. Li, M. Kang, J. Hou, and Y. Cao, "Mining node. js vulnerabilities via object dependence graph and query," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 143–160.

[22] T. Brito, M. Ferreira, M. Monteiro, P. Lopes, M. Barros, J. F. Santos, and N. Santos, "Study of javascript static analysis tools for vulnerability detection in node.js packages," *IEEE Transactions on Reliability*, vol. 72, no. 4, pp. 1324–1339, 2023.

[23] S. Geng, Y. Li, Y. Du, J. Xu, Y. Liu, and B. Mao, "An empirical study on benchmarks of artificial software vulnerabilities," *arXiv preprint arXiv:2003.09561*, 2020.

[24] J. Bundt, A. Fasano, B. Dolan-Gavitt, W. Robertson, and T. Leek, "Evaluating synthetic bugs," in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, 2021, pp. 716–730.

[25] A. Fasano, T. Leek, B. Dolan-Gavitt, and J. Bundt, "The rode0day to less-buggy programs," *IEEE Security & Privacy*, vol. 17, no. 6, pp. 84–88, 2019.

[26] J. Pewny and T. Holz, "Evilcoder: automated bug insertion," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, 2016, pp. 214–225.

[27] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "Redqueen: Fuzzing with input-to-state correspondence." in *NDSS*, vol. 19, 2019, pp. 1–15.

[28] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah, "{MOPT}: Optimized mutation scheduling for fuzzers," in *28th USENIX security symposium (USENIX security 19)*, 2019, pp. 1949–1966.

[29] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, 2018, pp. 2123–2138.

Appendix

```
let _babelInstTrace = (new Error().stack || "").
    toString().split("at␣")[1] || "";
    if (typeof _babelInstFs !== 'undefined') {
        try {
            _babelInstFs.appendFileSync("path/to
                /logfile", _babelInstTrace +
                _babelStringifyCircular(
                arguments) + '\n');
        } catch (_e) {}
    }
```

Fig. 5: Instrumentation code injected at the start of each function declaration.