Bullseye: Detecting Prototype Pollution in NPM Packages with Proof of Concept Exploits

Tariq Houis
Concordia University, Canada
tariq.houis@concordia.ca

Shaoqi Jiang
Concordia University, Canada
shaoqi.jiang@mail.concordia.ca

Mohammad Mannan
Concordia University, Canada
m.mannan@concordia.ca

Amr Youssef
Concordia University, Canada
youssef@ciise.concordia.ca

Abstract—**Prototype pollution is a critical security vulnerability in JavaScript, particularly in Node.js packages and applications, where attackers can manipulate the global object prototype and inject malicious properties into all objects that inherit from it. State-of-the-art static and dynamic approaches face significant limitations in detecting this vulnerability–both in terms of accuracy and efficiency. Static approaches struggle to recognize unexploitable vulnerabilities (e.g., due to missing code context with preventive mechanism), causing high false positives, besides suffering from scalability issues. Dynamic approaches have low false positives as they can access runtime information; however, due to low code reachability (resulting from the use of e.g., improper argument types/values), their false negatives could be high. In this paper, we present *Bullseye*, a fully automated dynamic analysis framework that delivers validated and scalable analysis of prototype pollution vulnerabilities in Node.js packages. Bullseye's novel approach combines broad entry-point coverage, context-aware exploit generation, and dual runtime validation oracles. We use the developer-provided inputs from a package's testsuites, and prototype pollution-related exploit inputs extracted from prior work. We then execute each entry point with its relevant exploit input candidates and observe the runtime for indications of prototype pollution. We analyzed 44,513 highly popular Node.js packages (with 10,000+ weekly downloads), and 5,879 packages with lower weekly downloads in less than 8 hours. We detected zero-day prototype pollution vulnerabilities in 290 packages, with no false positives. We responsibly disclosed all our findings with proof-of-concept exploit code to the respective package maintainers. We have also been assigned a total of 149 CVEs (as of July 22, 2025); among them, 66 have been made public, with 25 rated as critical, and 34 as high.**

I. Introduction

JavaScript is a highly dynamic language, allowing objects to be created, modified, and extended at runtime. This flexibility enables powerful programming constructs but also introduces significant security risks. Unlike statically typed languages where object structures are fixed at compile-time, JavaScript's objects are mutable, meaning that properties can be added or altered dynamically by modifying the object's prototype. This JavaScript feature is vulnerable to a relatively new type of vulnerability known as *prototype pollution*, first identified by Arteau [4]. It occurs when an attacker injects or modifies properties in an object's prototype, either by exploiting unsafe input handling, or by directly manipulating the special prototype properties (e.g., `__proto__` or `constructor.prototype`), and thus propagating the malicious changes throughout the entire application, impacting every object that inherits from the affected prototype.

Due to the widespread use of JavaScript, a significant number of studies have explored different approaches to identify prototype pollution, mostly focusing on the analysis of highly popular Node.js packages (as available on the NPM registry: npmjs.com), and applications bundling such packages; see, e.g., [4], [14], [12], [16], [15], [25], [18]. However, due to intrinsic complexity in JavaScript, most existing work based on static analysis faces significant challenges. For example, ODGen [17] (and similarly [16]) suffers from scalability issues due to path and object explosions inherent in its heavyweight abstract interpretation approach. As identified by Kang et al. [11], ODGen fails to complete the analysis of 50% of the tested NPM packages with an average 2K lines of code (LOC), which jumps to 90% for larger packages (with 60K LOC). Tools (e.g., [25], [15]) that rely on prototype pollution sink patterns, may miss vulnerabilities (i.e., false negatives), as the patterns used cannot cover all real-world vulnerable code. Such tools also suffer from false positives due to the overestimation of vulnerable sinks, which may be safeguarded by code that is not close to the sinks. In terms of dynamic analysis, Arteau's pioneering approach [4] uses a predefined list of exploit inputs (extended in [30]). However, while effective (with low false positives), such lists remain incomplete e.g., due to new vulnerability signatures being found. Also, these tools fuzz candidate functions for vulnerabilities, without considering a target function's input types, leading to poor code coverage.

To address scalability and validation issues of static analysis tools, and to overcome the limited code coverage of dynamic fuzzers, we design and implement a novel prototype pollution detection tool called *Bullseye*, which improves the detection rate (i.e., reduce false negatives) while avoiding false positives (i.e., no unnecessary reporting to developers). Bullseye automatically downloads a target package for testing,

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.230211
www.ndss-symposium.org

enumerates all its *entry points*[1] that serve as the interfaces to application developers, and leverages existing developer-provided testsuites for a guided execution. It performs real-time monitoring on each test case to check whether prototype pollution occurs, and provide proof-of-concept exploit code when a vulnerability is found.

Implementing Bullseye posed several challenges. For example, non-standard implementations across packages hindered the dynamic identification of usable modules using native import functions (e.g., `require`, `import`) alone. To address this, we adopted multiple import strategies to comprehensively identify all modules used within a package, including: support for different types of module declarations within package.json, covering e.g., CommonJS (CJS) and ECMAScript Modules (ESM) modules; dynamically imported modules; and user-accessible modules that are not explicitly indexed. After loading a package with all its modules–manifested as an object–Bullseye dynamically traverses this object to enumerate all the accessible entry functions, and comprehensively cover various types of export structures, e.g., entry points within complex exported objects, and dynamically generated exports.

After enumerating the entry points for a target package, we then face another set of challenges for executing them effectively and efficiently. To this end, we use a testsuite-guided mechanism for generating potential prototype pollution exploit inputs. Testsuites generally contain developer-provided example code to test a given package's functionality from an application context, specifically, the entry points. Example inputs from these testsuites should offer better code coverage as such inputs are specifically used for testing (compared to random/fixed inputs). For executing a given entry point, we directly use input values from testsuites, and specifically curated prototype pollution-related input fragments (mostly extracted from past work [4], [30], see Table I). Such guided input augmentation helps us uncover significantly more zero-day vulnerabilities compared to past work. If a package lacks testsuite, we rely only on the curated input fragments.

More specifically, for input extractions from testsuites, we take the abstract syntax tree (AST) representation of a test file, resolving import declarations, variable declarations, variable assignments, and function declaration nodes. We also locate the package's entry points by labeling the import identifiers, track them to where they are called (in the AST), and then record the function name and its resolved arguments. We then use pairwise testing (Czerwonka et al. [8]) to generate all possible combinations between the paired values from test inputs and exploit input fragments. Finally, we execute each entry point with its respective exploit input candidates, and check if newly introduced property in the global prototype chain appears at the runtime (indicating a vulnerability). Each test case is executed in a separate Node.js virtual machine (VM) so that individual test cases do not interfere with each other, and the failure of a single test case does not impede the rest of the analysis. For identifying the pollution attempt, we use two side-effect oracles, in contrast to the directly introduced property used in [4], [30], i.e., `if ({}.test==123)`. First, we recursively access the properties of the object's prototype, and search for the key or the value that matches our polluted property. For the second oracle, we use a differential check between a snapshot of the prototype chain before the exploit execution, and the one post execution. This oracle identifies more complex side-effects (not covered by the first oracle), such as introducing the property in unknown key-value pairs.

Overall, Bullseye's methodology introduces several key innovations in prototype pollution detection. (1) *Comprehensive module and entry point identification*: we implement comprehensive module import by enumerating module paths in package.json and non-indexed modules, and runtime detection of dynamic imports, while entry point identification is enhanced by identifying entry points in dynamic exports, complex export object, and class method-style entry points. (2) *Testsuite-guided input synthesis*: to address limitations in generating context-aware exploits, we dynamically generate exploit candidates by fusing project-specific valid inputs (from testsuites) and attack fragments (seed corpus) via pairwise testing. (3) *Robust prototype pollution monitoring*: we introduce two side-effect checking oracles: recursive prototype chain inspection and prototype chain differential checking with proxy-based setter interception, which enable the detection of pollution in nested or complex objects, providing a significant improvement over shallow checks. (4) *Isolated test case execution*: the use of Node.js's VM to execute each test case ensures its execution integrity (i.e., prototype properties cannot affect one another within a single shared environment), and the failure (e.g., timeouts, crashes) of a single test case does not affect the testing of other entry points/test cases within the same package.

Compared to state-of-the-art, our novelty lies in the following factors. In contrast to prior assumptions about dynamic analysis tools (as stated by e.g., [16]) missing many vulnerabilities due to low code coverage, we show that dynamic analysis can be reliably used to detect more vulnerabilities in NPM packages than existing tools (as evident from our evaluation). However, unlike prior dynamic approaches that treat prototype pollution detection as black-box fuzzing with fixed exploit lists, we introduce a testsuite-guided dynamic analysis that adapts exploit inputs to the specific argument structure of each entry point. Our methodology fuses project-specific valid inputs, extracted via AST analysis of developer-provided tests, with a curated and type-annotated seed corpus to generate context-aware exploit candidates—which enable us to trigger more vulnerable code paths compared to existing work. For false negative reduction, we further advance runtime detection by introducing dual side-effect oracles: recursive prototype chain inspection and differential prototype state comparison. Beyond the limitations of traditional dynamic detection, Bullseye identifies sink locations in a manner comparable to static analysis. Our design enables the detection of both shallow

[1] We treat exported functions from a package's public modules as entry points, all of which are accessible to an application developer. This enumeration may encompass a broader set of functions than what might be documented as formal entry points by the package developer.

and deeply nested prototype pollution vulnerabilities with zero false positives, while maintaining scalability to over 50,000 Node.js packages. To the best of our knowledge, this is the first work to demonstrate that combining testsuite-guided analysis with systematic entry point coverage and precise runtime side-effect detection can surface hundreds of previously unknown vulnerabilities at the ecosystem scale.

Main contributions and findings.

(1) We design and implement Bullseye, a testsuite-guided efficient dynamic analysis tool that detects prototype pollution vulnerabilities and generates reproducible proof-of-concept (PoC) exploits. Bullseye advances the state-of-the-art by increasing its reachability to more attack vectors (stemming from the combination of comprehensive entry point identification, input adaptation, and VM-based execution to avoid premature termination), and by avoiding false positives (through robust runtime side-effect monitoring).

(2) Our evaluation demonstrates that Bullseye significantly outperforms state-of-the-art tools in reducing both false negatives and false positives. We run Bullseye on 44,513 packages with 10,000+ weekly downloads, and 5,879 randomly chosen packages with under 10,000 weekly downloads—taking on average 0.51 sec/package of analysis time in a computer with an AMD Ryzen 2950X CPU. In total, we detected zero-day vulnerabilities in 290 packages, in 807 unique entry points. Many of these vulnerable packages are heavily used, e.g., 37 packages with 100k+ to 500k downloads/week, 13 packages with 500K+ to 1M downloads/week and 12 packages with 1M+ downloads/week.

(3) We responsibly disclosed all our findings, and as of July 22, 2025, we are assigned 149 CVEs, with 66 published CVEs: 25 are marked as critical, and 34 as high. In particular, the CVE (CVE-2024-39008) in the package 'fast-loops@1.1.3' (with 1M+ downloads/week) received a CVSS score of 10. Note that even though we identified 807 vulnerable entry points across 290 packages, we submit only one CVE per package even if a package contains multiple vulnerable entry points—to reduce the burden on the CVE program, and our manual submission effort. So far, 75 developers have responded to us and 31 of them confirmed fixing the reported vulnerabilities. We also received bug-bounties for 4 packages.

(4) We compared state-of-the-art tools (Arteau [4], Zhou and Gao [30], ODGen [17], Silent-Spring [25]) with Bullseye using past vulnerabilities, and zero-days discovered by Bullseye. Overall, Bullseye detected majority of the past vulnerabilities; in contrast, existing tools failed to uncover many of the zero-days, while reporting a significant number of false positives (specifically, ODGen and Silent-Spring). We also found zero-days in 9 packages from the Silent-Spring dataset (100 vulnerable packages).

(5) We will make our code and evaluation artifacts available at: https://github.com/Madiba-Research/Bullseye.

A video presentation of our work is available at: https://youtu.be/g1WBzvrdRjg.

II. Threat Model and Motivational Example

In this section, we provide our threat model and a motivational example highlighting the challenges in prototype pollution detection (see Appendix C for background).

A. Threat Model

We focus on Node.js packages deployed mainly in server-side applications. We assume these applications may utilize third-party Node.js packages potentially vulnerable to prototype pollution. Attackers aim to exploit these packages to alter the application's global state, thereby affecting other objects and services that depend on it. We assume that package and application developers are benign, and the attacker has no control over the package/application code; however, the attacker has access to at least one entry point in the package that can be triggered through interactions with the application, with the inputs controlled by the attacker, e.g., through a web application's client interface. While prototype pollution can lead to dangerous vulnerabilities such as arbitrary command execution, if the polluted data reaches sensitive runtime APIs (e.g., `execSync`, `execFileSync`), the specific consequences of prototype pollution are out-of-scope (but see e.g., [13], [12], [25], [18]). Our focus is uncovering exploitable prototype pollution vulnerabilities in Node.js packages.

B. Motivation and Challenges

To understand the challenges in prototype pollution detection, consider Listing 1, a simplified version of a utility function `merge` used for merging objects, from the package 'putil-merge' (github.com/panates/putil-merge).The function is vulnerable to prototype pollution (CVE-2021-25953) using the following exploit: `putil_merge(obj, payload, {deep:true})`. Where 'obj' is an empty object e.g., {} and payload is a prototype pollution payload, e.g., `{"__proto__":{"polluted":true}}`.

The function takes three arguments (`target`, `source`, and `options`), then iterates over the `source`'s properties. At line 6, 'srcVal' is assigned the injected property (i.e., the object '`{polluted: true}`'). At line 7, 'trgVal' is set to reference the prototype of `target` as the right side of the assignment becomes target['__proto__'], which retrieves the prototype of `target`. Next, the nested branches (lines 8-9) are triggered as payload and options.deep (from `{deep: true}`) are satisfied, after which the `merge` function is recursively called with 'trgVal' as the `target`, and 'srcVal' as the `source`. Consequently, the assignment process is performed on the new identifiers, assigning 'true' (from `{polluted: true}`) to 'srcVal' and 'undefined' to 'trgVal' (because unlike '__proto__', the key '`polluted`' does not exist at the `target`). Next, since the branch at line 8 is unsatisfied (as 'srcVal' is non-object), the execution moves to line 18, and 'srcVal' is assigned to `target`. Since target references to prototype, and key is undefined at the target, the construct (`target[key] = srcVal`) creates a new property with the key-value pair ‘polluted:true’ at target’s prototype, polluting the global prototype with this property.

While this function may appear to be a typical example of prototype pollution vulnerability, prior work failed to detect the vulnerability ([4], [30], [17]). To understand the reasons, we analyze the *exploit*[2] and the relevant test case (Listing 2), to identify the following challenges for modeling it.

(1) Inability to reach the entry point. Prior work [4], [30] fails to enumerate this entry point because of the nested function structure. Specifically, `merge` is defined with two labeled functions (`all`, `arrayCombine`), which caused the recursive entry point exploration loop to fall into these labeled functions, missing enumerating `merge` as an entry point.

(2) Unknown exploit input. Prior work [4], [30] relies on predefined exploit lists to detect prototype pollution, which contains fixed inputs commonly found in typical prototype pollution-vulnerable functions (as listed in Table IX, in the appendix). However, the function `merge` takes the object-typed argument ‘options’, which requires setting specific properties with boolean values (e.g., deep, clone) to trigger the vulnerability. The fixed input lists in prior tools [4], [30] exclude this function-specific argument, and thus fail to detect the vulnerability, specifically by not triggering the vulnerable branch (line 9), leading to the recursive call (line 12).

(3) Incomplete semantic modeling. ODGen [17] apparently struggles to model built-in property access functions. Specifically, ODGen fails to parse `getOwnPropertyNames` (line 2), hindering its ability to trace how the tainted data propagates towards the vulnerable line. We identified this limitation by replacing `getOwnPropertyNames` with a simple `for...in` loop for reading the properties, which enabled ODGen to recognize the vulnerability.

Our insight. For the `merge` example, we locate the related test case for the entry point (as shown in Listing 2), which includes the argument {`'deep:true'`}—without analyzing testsuite examples, such inputs cannot be efficiently generated. We then generate combinations based on the input (line 3), and our predefined seeds corpus (Table I), resulting in a list of exploit input candidates that combine `'{deep:true}'` (from the test input) with the exploit payload (from seed corpus). We could thus trigger and identify the vulnerable code. In short, we statically analyze testsuites to learn the input specifications of an entry point, and then use the specifications for generating our exploit input candidates. We then use the candidate inputs to execute the entry point and leverage the runtime to monitor the side-effects of the exploit execution.

III. Methodology

Overview. Fig. 1 presents our system architecture. Our tool, Bullseye, operates in three main stages: initial setup, exploit input generation and guided execution, and vulnerability summary refinement. In the initial setup stage (Sec. III-A), we prepare the target environment for analysis, which takes a

```
1  function merge(target, source, options = {}) {
2    for (const key of Object.getOwnPropertyNames(source)) {
3      if (options.filter && !options.filter(source, key))
4        continue;
5      const src = Object.getOwnPropertyDescriptor(source, key);
6      let srcVal = src.value;
7      let trgVal = target[key];
8      if (isObject(srcVal)) {
9        if (options.deep) {
10         if (!isObject(trgVal))
11           trgVal = target[key] = {};
12         merge(trgVal, srcVal, options);
13         continue;
14       }
15         if (options.clone)
16           srcVal = merge({}, srcVal, options);
17     }
18     target[key] = srcVal;
19   }
20   return target;
21 }
```

Listing 1. A code snippet from our motivation example

```
1    it('should deep clone function/class values to target', function() {
2      const a = {a: 1, b: 2, c: {a: Boolean}};
3      let o = merge({}, a, {deep: true});
4    });
```

Listing 2. The test case identified by Bullseye for the entry point `merge` in the package’s testsuite.

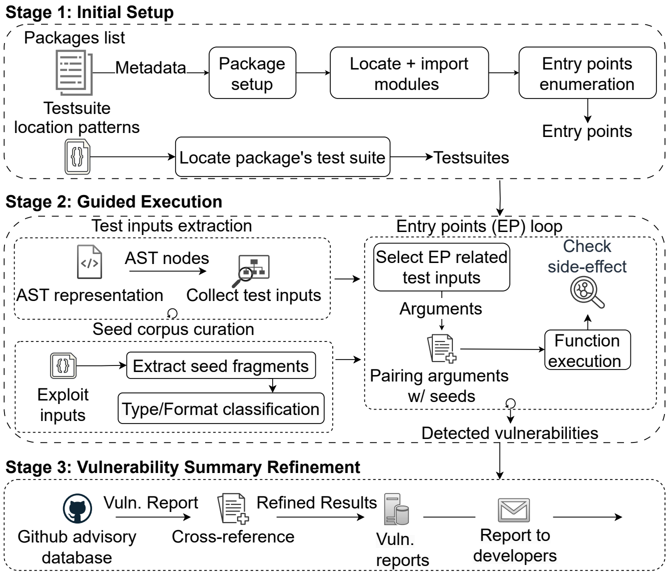

Fig. 1. Bullseye overview

[2] https://security.snyk.io/vuln/SNYK-JS-PUTILMERGE-1317077

list of Node.js packages for evaluation and processes each package iteratively. During each iteration, Bullseye locally installs the package using Node.js Package Manager (NPM) to ensure the package's codebase and necessary dependencies are available. The core task in this stage is to dynamically import the package and extract all its entry points. In the second stage (Sec. III-B), Bullseye dynamically executes entry points to inspect the presence of vulnerabilities, identify their locations, and determine the specific exploit inputs that trigger them. In this stage, we leverage the package's testsuites, which provide essential information about how each entry point function is invoked under various scenarios, and use such test inputs for crafting our exploit input candidates. With the crafted exploit inputs, Bullseye runs the target NPM package, and checks for indications of prototype pollution. In the final stage (Sec. III-C), we cross-reference the detected vulnerabilities with existing CVEs, and identify if the detected vulnerability is a potential zero-day or already confirmed. For zero-days, we automatically create a vulnerability summary report. For Bullseye's implementation details, see Appendix A.

A. Stage 1: Initial Setup

The setup process begins by reading a dataset of target packages. For each package, we use NPM CLI command to install its latest published version, including all dependencies listed in the package metadata file (package.json). We also clone the source code of the package from its repository link, obtained via the NPM Registry API (registry.npmjs.org) as we observed that some packages do not include testsuite files in their bundled NPM version. Next, we identify the package's testsuites and modules with entry points.

Testsuite identification. We locate all testsuite files within the Node.js package. While the package.json file may indicate these files under the "test" field, we noticed many packages do not follow this convention; variations include: the test file name matching the function (e.g., merge.test.js for testing the function `merge`), or the full path to the test file matching the function's path name (e.g., assign/object/merge.js for testing the function `assign.object.merge`). Moreover, to be distinguished from the package's modules, test files usually have a suffix or a prefix. We identified commonly-used keywords (e.g., test, spec, index), and how they are added to the test file's name, such as using a dash or dot (e.g., merge.test.js, spec-merge.js). Additionally, these keywords could also be in a parent directory containing the test file (e.g., spec/merge.js). To locate common path patterns, we first select 10 packages from a set of 100 vulnerable Node.js packages (from Silent-Spring [25]), and create path patterns to locate their test files. Then we apply these patterns on the rest of the packages, and select another 10 packages for manual analysis where no testsuites were detected, and update our list of patterns (if new patterns are found). We continue this process until we curate a comprehensive path patterns that can locate all the test files in the 100 Node.js packages. Then, we use the glob library [28] to run the patterns on the package's directory to fetch the matched test files; see Table VIII (in the appendix) for these patterns.

Comprehensive module import. While the native importing functions (e.g., `require`, `import`) can be used to import a package, we noticed that, in many cases, importing cannot be fully accomplished with a uniform pattern (e.g., solely rely on `require`). These cases, as discussed below, if not handled properly, may significantly affect the import of modules under test (i.e., leading to low code coverage).

(1) Different module systems specify their own import functions. In most cases, a package supports only one module system, such as CommonJS or ES modules. This requires using the correct import method: `require` for CommonJS and `import` for ES modules. However, some packages offer universal support for multiple module systems. They provide different versions of the same module, distinguished by file extensions (e.g., module.cjs.js, module.esm.js, module.es2015.js). These versions are then defined in package.json. In this scenario, relying on a single import method may prevent detection of potential vulnerabilities in other versions.

To support different module systems, we inspect the package.json file for keys such as 'exports.import', 'exports.require', 'module', 'main', and 'jsnext:main'. Next, we use the appropriate native import function for each module type, e.g., `require` for modules under 'exports.require' and 'main', and `import` for those under 'exports.import', 'module', and 'jsnext:main'. Additionally, for universal packages that export functions with the same name, we assign a property to label the type of newly imported object (e.g., esm: lib.esm, cjs: lib.cjs, es6: lib.es6). These labels are then used to cover different versions of the same module in the package.

(2) Some packages do not explicitly index modules in package.json. These modules can still be accessed with relative paths (e.g., `var lib=require('lib/module.js')`). For such cases, we recursively traverse the internal modules of the package for JavaScript files, and import each module.

(3) Some modules are imported implicitly. Some packages do not require assigning the imported module to a variable. Instead, they use a bootstrap mechanism that dynamically creates a global namespace object when the package is imported. For this case, we store a copy of the global object before and after we import the package. Then, we compare the difference of both copies, which shows the newly imported package in the global object (if any).

Entry point identification. To extract the entry points from an imported object, we iterate over the properties of this object, storing all function-typed properties. We apply the loop statement *for..in*[3] to perform this process (e.g., `for (const fn in lib)`). During this process, we discovered that the structure of the imported objects varies. To access these objects and retrieve their entry points, we execute two enumeration processes. One uses the *for..in* statement, and the other uses `Reflect.ownKeys`. Each process returns the first-

[3] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in

level properties of the imported object. Next, we recursively enumerate each of these properties. Each time we inspect the property type to decide whether to store this property (if it is a 'function-typed'), or further recursively enumerate it (if it is 'object-typed'). We encounter the following issues while directly traversing the properties of the imported object: (i) in some cases, enumerating function properties may reveal additional nested functions; (ii) some functions are only accessible as properties of a class.

To address (i), after applying `Reflect.ownKeys`, we check if the a `function-typed` value can be further explored before we finally store it. To address (ii), we check if the property has the following built-in class methods: `get`, `set` and `has`. The presence of these methods indicates the value type of class, which potentially contains entry points as properties of a class. If so, we enumerate its methods by accessing its prototype property (e.g., $Reflect.ownKeys(prop.prototype)$).

B. Stage 2: Exploit Input Generation and Guided Execution

In this stage, Bullseye examines if there is any entry point in a given Node.js package that can trigger prototype pollution vulnerability. To generate exploit input candidates, we need prototype pollution-related values and the function-specific argument structures. Thus, we enumerate a list of input fragments (seed corpus), which are values that can effectively trigger prototype pollution vulnerabilities as found in prior work [4], [30]. However, prior work uses these values in fixed inputs, disregarding the entry function's signature/argument structure. We thus leverage the package's own testsuite to identify test inputs for each entry point, and determine expected argument structures and values. We then mutate this argument structure with our predefined fragments (seed corpus).

Seed corpus curation. We prepare a list of prototype pollution fragments to serve as seeds for generating exploit input candidates. First, we reviewed prior work [4], [30], specifically the exploit input list (see Table IX in the appendix), and selected 20 fragments that we could use as primitive values for generating exploit input candidates. We curated these fragments in subsets, each of which is tagged with a data type (e.g., array, object, string). Moreover, we defined special data types (e.g., N-array, S-string) to address the types that are often found in prototype pollution payloads. For example, some string inputs in prototype pollution payloads come with a specific pattern: slash, dot, colons, or bracket (e.g., "/obj/prop", "obj.prop", "obj:prop", "obj[prop]"). To properly generate exploit input candidates, we need to use the right string based on the matched one we found in the testsuite. Thus, we distinguish each of these types with special names respectively: S-string, D-string, C-string, A-string. We also define special array type: 'N-array' for the exploit inputs that take multi-dimensional arrays. This special type covers inputs in the formats such as [[FRAG1],FRAG2], [[FRAG1,FRAG2], VALUE]. We list our curated seeds in Table I.

Note that we do not include any values with the type of integer or boolean as prototype pollution fragments, because they cannot be exploited for prototype pollution directly. Furthermore, we do not take the constructor.prototype fragment (found in Table IX, in the appendix, as BAD_JSON2 at number 17, 18, 19, and string values in 21, 23, 30, 33, 38, 42), as we noticed the same effect of using __proto__ and constructor.prototype, which causes redundant vulnerability triggers. Similarly, we exclude file fragments (e.g., 43, 44 in Table IX, in the appendix), as they did not yield results when tested on 60,000 Node.js packages [30].

We also expanded the list with missing fragments by prior work (such as the fragments 6, 13, 15, and 18 in Table I). These fragments allow discovering vulnerabilities that prior work cannot find. For example, fragment 6 triggers a new vulnerability at 'accessors/set.js:37' in the package '@cahil/utils@2.3.2', because the vulnerable code requires an array with three elements to access the enclosed conditional branch.

No.	Type	Seed Input
1	object	{}
2	object	{}.__proto__
3	object	{"__proto__.pollutedKey":"pollutedValue"}
4	object	JSON.parse('{"__proto__ ":{"pollutedKey":"pollutedValue"}}')
5	array	["__proto__ ", "pollutedKey"]
6	array	["__proto__ ", "pollutedKey", "pollutedValue"]
7	N-array	[["__proto__ "], "pollutedKey"]
8	N-array	[["__proto__ "], "pollutedKey", "pollutedValue"]
9	N-array	[["__proto__ ", "pollutedKey"], "pollutedValue"]
10	N-array	[["__proto__ "], ["__proto__ "], "pollutedKey"]
11	string	"__proto__ "
12	S-string	"/__proto__/pollutedKey"
13	S-string	"/__proto__/pollutedKey=123"
14	D-string	"__proto__.pollutedKey"
15	D-string	"__proto__.pollutedKey.pollutedValue"
16	D-string	"__proto__.pollutedKey=123"
17	C-string	"__proto__:pollutedKey"
18	C-string	"__proto__:pollutedKey=123"
19	A-string	"__proto__ [pollutedKey]"
20	A-string	"__proto__ [pollutedKey]=123"

TABLE I
BULLSEYE'S SEED CORPUS, ORGANIZED IN DATA TYPES AND SPECIAL FORMATS.

Test input extraction. In this step, we extract all test inputs that relate to our entry points. We use Abstract Syntax Tree (AST) to locate the entry point of interest and extract all related test cases. The tree representation of the test file's code facilitates tracking the data flow from one variable or function call to another. For instance, assume that node A represents the declaration of a variable p (e.g., `var p`); node B represents the assignment of a value to p (e.g., `p = 1`); and node C represents the use of p as an input to a function call (e.g., `fun(p)`). Therefore, in the tree structure, edges connect these three nodes sequentially. If the function `fun(p)` is an entry point, we can locate it as node C, and determine the type and value of variable p, by tracing node B pointing to C, and further node A pointing to B.

We start from the root node and traverse all nodes in the tree. During the traversal, we examine function call nodes (node C) and check if this function is an entry point. If the called function is an entry point, we backtrack all nodes pointing to the node representing the call to the entry point, as these nodes represent the input variables (node B) of the entry point function. Additionally, for each input variable node (node B), we continue backtracking until the variable is initially declared (node A). By backtracking through these nodes, we can determine how an entry point's input variable was declared, how it was updated, and its value is when the entry point is called.

Entry points identification in testsuites. When analyzing testsuites, we found that other than the package's entry points, testsuites can also import and execute functions from other modules (e.g., calling the `assert` function to compare results). Therefore, we need to ensure that we only track the entry points belonging to our target package and ignore unrelated functions. To achieve this, we collected the path patterns of import modules from the testsuites of 100 Node.js packages (from the same packages mentioned in Sec. III-A). We curated these patterns into a list (see Table VII). During the testsuite processing, if we identify an import path matching one of these patterns, we label the assigned identifier to track the point where the related entry point is called. For example, consider that a function is called in a testsuite as a property of the imported identifier (e.g., lib.fun(p)). We identify the importing node in the AST (e.g., `import lib from "../index.js"`) containing one of our curated import paths. We label this identifier (lib), and traverse the tree until we find the call expression node that matches the one we labeled (lib.fun(p)); we then store the values of test input's arguments. These records are further utilized to generate the exploitable input candidates.

Exploit input generation. For each test input of an entry point, we synthesize a batch of input candidates. Each candidate consists of a valid sequence of arguments for the entry point. In each candidate, one of its arguments is replaced by a value from Table I, serving as an effective payload to trigger potential prototype pollution. Recall that we record the data types and values for each test case, from which we generate a batch of candidates. Given a test case with a sequence of arguments, we check the type of each argument. If the type of an argument matches the type of an entry in Table I, the original value of the argument is replaced by the value in that entry. We count this new sequence of arguments, with one value replaced, as a candidate. We generate a batch of candidates in a *pairwise* manner [27] to cover as many potentially effective payloads as possible. We explain this more using our motivating example.

In Listing 2, a test case with three arguments ({}`, a,` {`deep: true`}) is served into entry point `merge`, where 'a' is an object defined as `const a =` {`a: 1, b: 2, c:` {`a: Boolean`}}. We first identify the type of first argument as object, and then find all entries in Table I that are also typed as 'object' (# 1, 2, 3, 4). Then we replace the original argument value with the seed input values of these entries to generate our candidates. In this case, we have four candidates targeting the first argument: (`{}, a, {deep: true}`), (`{}.__proto__, a, {deep: true}`), (`{"__proto__.pollutedKey": "pollutedValue"}, a, {deep: true}`) and (`JSON.parse('{"__proto__": { "pollutedKey": "pollutedValue"}}'), a, {deep: true}`). Similarly, another four candidates will be generated by replacing the value of the second argument in the original test case, and another four of the third argument. We thus have a batch of 12 candidates mutated from this original test case for the entry point `merge`. If there is another test case for `merge` in the testsuite, a new batch of candidates will be generated based on that test case. In the end, we execute the entry point (`merge`) with all candidate inputs.

Guided execution. After collecting the entry points and generating the exploit input candidates, Bullseye executes each entry point with its relevant exploit input candidates and actively monitors for signs of prototype pollution. This process ensures precise detection by observing the runtime's behavior in real-time and identifying the vulnerable code that enables prototype pollution. However, using a shared environment between Bullseye and the tested function is risky, as we noticed cases where Bullseye stopped working because of an endless loop invoked by the tested function. This is one of the drawbacks of prior tools [4], [30], as they cannot complete the scan on a dataset if any of the tested functions invoked an endless loop. We addressed this challenge by using Node.js's VM module,[4] which can be used to trigger a timeout for long-processing functions. We set the timeout to 100 ms for executing an entry point against each exploit input candidate, which we found to be enough for executing any function without infinite loops.[5]

An obvious sign for prototype pollution is the inheritance of the injected property from any object in the running application, including the empty object. However, as we observed, the injection can occur in other places. (1) The polluted property can be added with an empty value (e.g., $\{polluted : \text{“”}\}$), or under another object (e.g., $\{someObj : \{polluted : true\}\}$). (2) The key and value in the injection combine as a key name of an object (e.g., $\text{“}polluted = true\text{”} : \{\}$), making a simple check ineffective (e.g., $\{\}.polluted$). To solve these cases, we introduce two side-effect checking oracles: recursive prototype chain inspection and prototype chain differential checking to detect pollution in nested/complex objects. In the first oracle, we recursively access the prototype chain, looking for the injected property by matching our predefined key-value pairs with each property we read from the prototype chain. Next, the second oracle is used to detect the side-effect even if the key-value pair of the polluted property is mutated by the target function. We compare the prototype chain before and

[4] https://nodejs.org/api/vm.html

[5] We found only 6 packages with such loops: gammautils@0.0.81, locutus@2.0.11, mout@2.0.0-alpha.1, nis-utils@0.6.10, node-forge@0.9.0, nodee-utils@1.2.2. E.g., the function *shuffle* in gammautils@0.0.81 has the condition `while (0 !== currentIndex)`, where `currentIndex` is the array length of the argument. If the supplied argument is not an array, `currentIndex` becomes `NaN`, consequently the while loop is always true.

after the injection attempt to reveal the changes occurred due to the pollution. Specifically, we clone the global prototype (e.g., using `Object.getPrototypeOf`) before and after we execute the exploit. Then, we read the property names from each clone (e.g., `Object.getOwnPropertyNames`). After having both arrays of properties, we apply Array's `find` method to find the newly added property.

Beside detecting prototype pollution side-effects, we also attempt to identify the vulnerable code line. We use Proxy [23] to intercept the pollution attempt by customizing the `set` handler to detect the polluted property. Specifically, we modify the `set` handler of an empty object's prototype. Then, we replace the empty object at the exploit input under execution with the proxied object. Upon the entry point execution, if the prototype pollution occurs, the runtime returns to *set* handler and we check if the property to be added is __proto__ or *polluted*. In that case, we use the runtime's call stack to get the last executed line of code before Bullseye, which represents our sink line. Note that this approach only works when there is an object argument in the exploit candidate in order to trigger the `set` handler on the pollution attempt.

C. Stage 3: Vulnerability Summary Refinement

In the final stage, we generate a vulnerability summary based on the results of the guided execution. We include the location of vulnerable sinks (if available), the package's entry points, and the executable proof-of-concept exploits in detail. We also compare these vulnerabilities with previously disclosed ones. Specifically, we retrieve publicly disclosed vulnerabilities for a target package as identified by their CVE IDs (if any). We use the GitHub Advisory Database[6] for CVE data, through the Octokit API client [9], querying both the package name and the CWE ID CWE-1321, which corresponds to prototype pollution vulnerabilities. The API returns CVEs related to prototype pollution for the target package, including detailed descriptions that may identify affected components (e.g., function names, code snippets, line numbers, or file paths). We cross-reference these details with the recorded entry points and sink locations, and add the relevant CVE IDs (if any) to the final vulnerability report.

We determine whether a finding qualifies as a zero-day as follows. We automatically de-duplicate by comparing the entry points of discovered vulnerabilities with those in known prototype-pollution CVEs, when a CVE includes entry point details. Otherwise, we conservatively assume that the CVE might cover our finding and treat it as a duplicate. That is: a finding is classified as a zero-day only if, for the affected package, no CVE shares the same entry point as we found, or if no prototype pollution-related CVEs exist for the package. This conservative strategy minimizes duplicate reports at the expense of potentially missing some true zero-days.

IV. Evaluation

We focus on answering key research questions that assess Bullseye's ability to detect known and unknown/zero-day prototype pollution vulnerabilities. **RQ1:** How effective is Bullseye in detecting previously reported prototype pollution vulnerabilities? How does it compare with existing tools? **RQ2:** How effective is Bullseye in uncovering zero-day prototype pollution vulnerabilities in the wild? **RQ3:** How effective are existing tools in detecting zero day vulnerabilities uncovered by Bullseye? **RQ4:** How effective are various components of Bullseye in detecting zero day vulnerabilities?

A. Experimental Setup

Environment. We performed our evaluation on a physical machine running Ubuntu 22.04, equipped with a 16-core AMD Ryzen Threadripper 2950X CPU (released in 2018), 64 GB of RAM, and 8 TB of SSD storage. We employed Docker version 24.0.5 for our analysis. The containerization ensures that our physical device remains unaffected by the Node.js packages under inspection. Meanwhile, it isolates the process of running each package, preventing any version conflict for common dependencies. We limit the maximum number of containers running in parallel to 64 (for maximum utilization of our CPU).

Datasets. To evaluate the effectiveness and accuracy of our system against existing tools, we use the datasets provided by the authors of those tools. We also test Bullseye on real-world Node.js packages, focusing on widely used packages, with at least 10,000 weekly downloads, which we can get by querying the package's metadata from the NPM registry API. However, because of the API rate limit in NPM,[7] we only fetch the maximum allowed packages every day and save only the ones with 10,000 weekly downloads, repeating this process every day. Eventually, we ran the script continuously for three months (June-August, 2024), curating a dataset of 44,513 packages. We also tested randomly-chosen packages (starting from early 2024) irrespective of their download rates, at various stages of our tool development, and eventually selected 5,879 of these packages for evaluation, all of which had a weekly download rate less than 10,000. These two datasets allowed us to understand prototype pollution vulnerabilities in packages with different popularity levels.

Baseline tools. We benchmark Bullseye against prominent dynamic and static analysis approaches. For the dynamic analysis baseline, we use two fuzzing tools from: Arteau [4], and Zhou and Gao [30]. For the static analysis baseline, we use the tool proposed by Li et al. [17], ODGen, which is a general-purpose tool for detecting JavaScript vulnerabilities including prototype pollution. The second static baseline is Silent-Spring [25], which uses four different CodeQL queries based on the scope for scanning the package. To have a fair comparison, we choose "Priority queries/Exported Functions" which is the most relevant to the scope of Bullseye, as we identify a vulnerability by finding the injected property, and we only test exported functions.

[6] https://github.com/advisories

[7] https://blog.npmjs.org/post/164799520460/api-rate-limiting-rolling-out.html

B. RQ1: Detection of Past Vulnerabilities

First, we benchmark Bullseye against Node.js packages previously reported with prototype pollution (as found in our baselines). We compare our results based on the vulnerability details provided by the baseline tools, e.g., the entry point function, the path or line number of the sink. If the vulnerability details are partially provided, such as a sink location, we match it with the sink location reported by Bullseye. For the results where Bullseye does not provide a sink line, we manually debug the generated exploit by Bullseye to find the sink line and compare it with the baseline.

Arteau [4]. We evaluate Bullseye against the 15 vulnerable packages listed by Arteau (including the package version and the vulnerable entry points). Bullseye was able to detect prototype pollution in all packages. In addition to all the vulnerable entry points from Arteau's list, Bullseye found 3 new vulnerable entry points (`pick`, and `updateWith` in 'lodash@4.17.4', and `clone` in 'deap@1.0.0').

Zhou and Gao [30]. From the 65 packages listed by the authors with vulnerabilities, only 6 packages come with a CVE ID.[8] Thus we test Bullseye against these packages, as we can use the details of these vulnerabilities from their CVE details for comparison. All vulnerabilities were detected by Bullseye.

ODGen [17]. We test Bullseye against 19 packages with prototype pollution vulnerabilities listed by the authors. For comparison, we used the benchmark (for the same vulnerabilities) from Silent-Spring [25] that includes the sink lines and the PoC files that trigger the injection for each package. If the exploit case generated by Bullseye does not contain a sink location, we ran the entry point with the same exploit case in debug mode, and manually traced the execution until it reached the same sink location from the benchmark list. Bullseye detected 26/38 vulnerable sink locations (missing 12, with no false positives). We manually checked all 12 false negatives (FNs), and found that infeasible attack vectors and complex exploits are the primary reasons for missing them (detailed in Appendix B).

Silent-Spring [25]. We evaluate Bullseye against Silent-Spring using the 100 of Node.js packages listed by the authors. For vulnerabilities without sink locations, we checked them manually as we did for comparison with ODGen. Table II summarizes our results. After manually checking the installed packages and executing all PoC files, Bullseye is able to detect 92/134 vulnerabilities. The primary reasons for FNs include complex exploits, infeasible attack vectors, and unknown payload patterns (detailed in Appendix B). We also found 4 apparent false positive cases. We tested the PoC (CVE-2020-28271) for the package 'deephas@1.0.5' and noticed that the exploit only affects the target object but not the global prototype. For the package 'dot-object@2.1.2' (CVE-2019-10793), we verified the disclosed exploit,[9] and found that the flagged sink locations in the ground truth are unexploitable.

Bullseye also uncovered 24 unknown sink locations in 12 packages in the Silent-Spring ground truth dataset. 17 of these sink locations in 9 packages are confirmed zero-days (7 remaining sinks in 4 packages have already been reported in CVEs). We contacted developers of these 9 packages with our vulnerability reports. 5/9 packages remained unpatched in their latest versions (as checked on July 22, 2025).

Tool	Total sink	TP	FN	FP	Duration (sec)
Bullseye	134	92	42	0	371
Silent-Spring	134	112	22	113	1850

TABLE II
OVERALL DETECTIONS RESULTS FOR THE 134 VULNERABLE SINK LOCATIONS FROM THE SELECTED 100 PACKAGES.

C. RQ2: Uncovering Zero-Day Vulnerabilities

We run Bullseye on 44,513 packages with at least 10,000 weekly downloads (labeled as high-DL packages), and 5,879 randomly chosen packages (with less than 10,000 weekly downloads, labeled as low-DL packages). In total, we detected zero-day vulnerabilities in 290 packages (250 from high-DL and 40 from low-DL packages), in 807 unique entry points (655 from high-DL and 152 from low-DL packages), and a total of 1,172 exploitable test cases (950 from high-DL and 222 from low-DL packages). Note that each entry point represents an independent attack vector; to avoid overwhelming the developers, we share only one test case for each entry point, and to reduce CVE reporting (a manual process via cveform.mitre.org), we submit only one CVE per package (by grouping all vulnerabilities in a package if it has multiple ones). Many vulnerable packages are widely used; see Fig. 2.

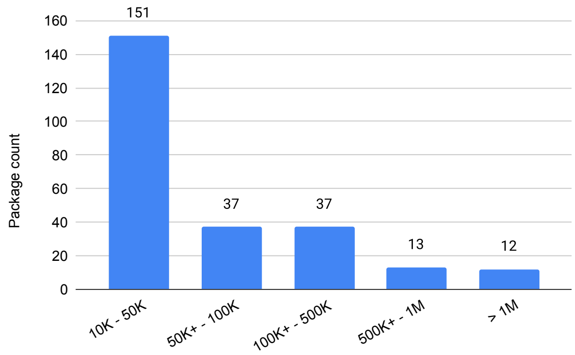

Fig. 2. Distribution of downloads for 250 vulnerable packages (from the high-DL 44,513 packages).

We responsibly reported all our findings to the respective package maintainers, and submitted for CVEs. As of July 22, 2025, we are assigned 149 CVEs, with 66 published CVEs. Out of the published CVEs, 25 are marked as critical, and 34 as high, and 7 as medium; see Table VI (in the appendix) for the critical and high severity CVEs. From the 250 high-DL vulnerable packages, 118 CVEs have been assigned; 35 of

[8] They are: 'safe-eval@0.4.1', 'flatnest@1.0.0', 'collection.js@6.7.11', 'rangy@1.3.1', 'progressbar.js@1.1.0'.

[9] https://security.snyk.io/vuln/SNYK-JS-DOTOBJECT-548905

them have CVE scores published (7 rated as critical, 26 high, and 2 medium). From the 40 low-DL vulnerable packages, 31 CVEs have been assigned and made public with severity scores (18 rated as critical, 8 high, and 5 medium).

Overall, our feedback received positive responses, and apparently, led to the resolution of security issues in several packages. As of July 22, 2025, 75 developers have responded to us and 31 of them confirmed fixing the reported vulnerabilities. Additionally, one developer noted that the vulnerable entry point was not mentioned in the official documentation, suggesting it should not be used by developers (even though it is a valid entry point). Some developers did not fix the bug for the package version we reported, but fixed it in a new version (assuming the older version should not be used by application developers). For some old packages, the developers refused to fix the bugs even though these packages have high download rates. We also tracked the status of vulnerable packages maintained by developers who did not respond within 90 days after our email notification. We found that 11 of these packages appear to be unmaintained (i.e., no updates for more than a year, no active response to public issues on GitHub). Notably, for one package, the developer archived the GitHub repository and marked it as deprecated one month after we sent our notification email. In addition, four other packages were fixed after the relevant CVEs were publicly disclosed.

As we check the latest versions of the vulnerable packages again with Bullseye (July 22, 2025), we found that 129/290 packages were no longer vulnerable. See Fig. 3 for both the CVSS severity (left bars) and weekly download counts (right bars) of 23 high-DL packages.

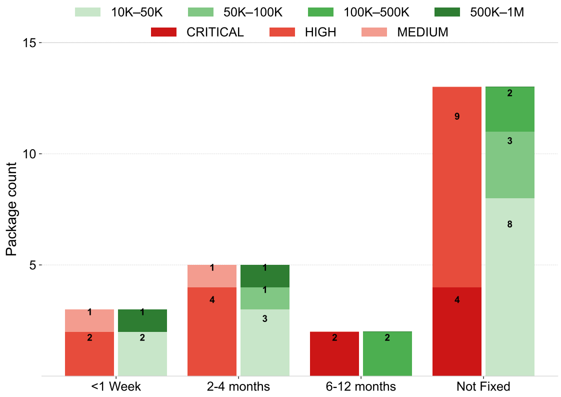

Fig. 3. Patching status distribution of 23 high-DL packages by CVSS severity ratings (left bar), and weekly DL counts (right bar). To determine if/when vulnerabilities were fixed, we tested all versions of these packages released after our reporting.

To measure the performance efficiency of Bullseye, we recorded the running time of each package when we tested both low-DL packages (5,879) and high-DL packages (44,513). Bullseye employed a parallel execution mode, allowing multiple packages to be tested simultaneously. In our experiments, we excluded download and installation time, and used 64 containers to run in parallel, each with one test package. We first evaluated the 5,879, where all packages completed testing within 50 minutes. Subsequently, we tested the packages in 44,513, which completed in 6.3 hours. With Bullseye's parallel execution, it took an average of 0.51 seconds per package. For individual packages, Bullseye takes an average of 32.38 seconds/package for evaluation (min. 2 seconds, max. 531 seconds, with a standard deviation of 20.63 seconds). In addition, we grouped these packages into 30-second intervals; see Fig. 4. Overall, 98.2% of the packages are completed within 1 minute, and 99.8% within 3 minutes.

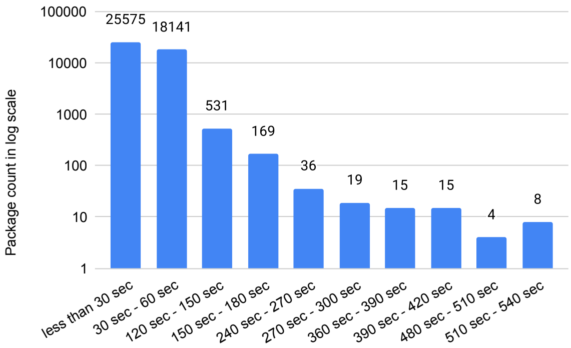

Fig. 4. Distribution of packages based on their execution completion times for the high-DL dataset (44,513 packages). Note that we omit the time ranges with package-count zero.

D. RQ3: Detection of Our Zero-Days by the Baselines

For baseline comparison, we randomly selected 40 packages from 137 zero-day packages where we also have the vulnerable sink information; see Table III for results summary.

For dynamic baselines, as they do not report sink lines by default, we embedded our proxy detector in both (Arteau [4], Zhou and Gao [30]), to get the sink lines. They missed almost half of the vulnerabilities, a strong justification for our use of testsuite-guided input augmentation—instead of blindly using the fixed exploit inputs as in the baselines. On the other hand, the dynamic baselines detected a few sink locations that Bullseye could not. For the two sink lines (i.e., vis-util.js: 3255, 3257) found by both baselines, the related entry point ('deepObjectAssign') is passed as an argument to the test function (e.g., $test(deepObjectAssign, () => \{//...\})$, which currently Bullseye cannot parse. For the sink location at (xe-utils/set.js:53), Zhou and Gao [30] detect this vulnerability with an input that has no matching format in the testsuites, specifically the two-dimensional array in the second argument (e.g., $(\{\}, [[\_\_proto\_\_], \text{“}test\text{”}], \text{“123”}, true)$. In the testsuites, however, all three relevant test cases have one-dimensional array at the second argument. For the two sink lines in cloneextend/index.js (120, 123), the entry point of interest ('extenduptolevel') has no matching test cases in the package.

For the static baselines, we note that ODGen has overall low detection rate (8/87 zero-days). Silent-Spring, on the other hand, has higher detection rate but potentially more false positives (while still missing 53/87 zero-days). Since static baselines do not generate exploits to validate the findings, we are not sure about the 55 sink lines in ODGen and 91 sink lines in Silent-Spring for which Bullseye did not provide a PoC; we label them as unknown or potential false positives.

Possible failure reasons in baselines. We found that both Silent-Spring and ODGen often fail to complete the analysis on complex packages. ODGen's failures are largely due to limited code coverage and call graph imprecision, while Silent-Spring struggles with dynamic JavaScript features such as bracket-based function calls and dynamic property accesses. These observations align with the findings by Kang et al. [11], and Zhou and Gao [30], which showed that ODGen's false negatives stem from its exponential growth of analysis nodes and scope mismatches, and that Silent-Spring's reliance on CodeQL makes it ineffective for certain dynamic language constructs.

Additionally, we found another important weakness related to incomplete modeling of built-in JavaScript functions in ODGen. For vulnerable code locations detected by Bullseye but missed by the baselines, we manually replaced the critical lines with semantically equivalent alternatives. For example, in Listing 1, we replaced $Object.getOwnPropertyNames(source)$ in the property access loop with a direct property access loop, after which ODGen, Arteau, and Zhou and Gao could detect the vulnerability, indicating that the missed detections are due to incomplete modeling of certain built-in JavaScript functions (not the vulnerability logic itself).

Tool	Detected sink	FP/Unknown	TP	FN	Duration (sec)
Arteau	45	2	43	44	337
Zhou and Gao	52	5	47	40	498
ODGen	63	55	8	79	28080
Silent-Spring	125	91	34	53	2938
Bullseye	**87**	**0**	**87**	**0**	**875**

TABLE III
BASELINES' DETECTION PERFORMANCE ON OUR SELECTED ZERO-DAYS (WITH A TOTAL OF 87 GROUND TRUTH SINK LOCATIONS).

E. RQ4: Effectiveness of Bullseye Components

The design of Bullseye consists of multiple components. Recall that in the initial stage, the loading component traverses all modules contained in the package, and for each discovered module, the enumeration component further enumerates all exported functions as potential entry points. Then in the guided execution stage with input augmentation, the test case generation module first examines the testsuites in each module and synthesizes pairwise test cases that are specifically targeted towards the identified entry points. Each of these generated test case is then executed within the VM by invoking the entry points. Finally, the side effect monitor observes any side effects caused by prototype pollution. In our ablation study, we systematically analyze each component in the initialization and guided execution stages, by removing/substituting each with a related variant from previous work (primarily Arteau [4]) to analyze its impact on the overall detection capability.

1) Components in Initial Stage: In this scenario, we explore how the initialization of a package under inspection affects Bullseye's results. Both the loading and the enumeration components may influence the detection outcomes, as they determine the number of modules discovered, as well as the number of entry points used for testing within each module.

Therefore, we design the following experiment, in which we modify Bullseye to create the following variants: (1) in Bullseye$_{\text{ArteauLoad}}$, we replace only our loading component with that of Arteau. (2) in Bullseye$_{\text{ArteauEnum}}$, we replace only our entry point enumeration component with that of Arteau. (3) in Bullseye$_{\text{ArteauInit}}$, we replace both components (loading and enumeration) with the corresponding ones from Arteau. Note that Arteau's tool is not modularized and has a high degree of implementation coupling, which makes direct component replacement difficult. We thus adapted Arteau's loading and enumeration strategies within our system to the extent possible. Additionally, to gain more insights into how our loading and enumeration components enhance our test coverage, we performed another experiment: for the vulnerable packages discovered by each variant, we executed only the initialization stage and recorded the number of modules and entry points reached by that variant.

The results from these variants are summarized in Table IV. In both the high-download and low-download datasets, Bullseye outperforms other variants in terms of discovering vulnerable packages and identifying exploitable entry points. We also noticed an unexpected result where the combined variant Bullseye$_{\text{ArteauInit}}$ yielded more exploitable entry points than the enumeration-only variant Bullseye$_{\text{ArteauEnum}}$. To understand this discrepancy, we examined Arteau's enumeration logic in detail and found that it stems from a compatibility issue: Arteau's function enumeration cannot effectively traverse modules that we comprehensively collected from our package loading component. In terms of the overall enumerated modules and accessible entry points, our loading and enumeration components also produced the best outcomes.

2) Components in Guided Execution: In this scenario, we focus on the input generation and testing strategies used in Bullseye. Recall that our methodology includes pairwise test case generation, VM isolation, and side-effect monitoring. To figure out the contribution of these components, we create the following variants of Bullseye. (1) In Bullseye$_{\text{NoVM}}$, we remove the VM that isolates the execution of each test case. (2) In Bullseye$_{\text{NoPW}}$, we remove the pairwise generation component and rely solely on the fixed inputs used in prior work. (3) In Bullseye$_{\text{ArteauSE}}$, we replace only the side-effect detection with Arteau. (4) In Bullseye$_{\text{NoVM, NoPW}}$, we remove both VM and pairwise generation, while retaining our side effect monitor. (5) In Bullseye$_{\text{NoPW, ArteauSE}}$, we remove pairwise generation and replace our side-effect monitoring with Arteau. (6) In Bullseye$_{\text{ArteauFuzzy}}$, we remove all our three components, and keep Arteau's fuzzing execution. Table V summarizes our results. It is evident that each of the new components of

Bullseye significantly enhances the detection capability, i.e., the use of more components leads to the detection of more vulnerable packages.

Variant	# Vuln. Pkg	# Exp. EP	# Module	# AEP
Bullseye	290	818	1353	154889
Bullseye$_{\text{ArteauLoad}}$	279	518	298	102422
Bullseye$_{\text{ArteauInit}}$	193	225	298	12870
Bullseye$_{\text{ArteauEnum}}$	58	68	1353	866

TABLE IV
COMPARISON OF BULLSEYE VARIANTS MUTATED IN THE INITIALIZATION STAGE (FOR ALL 290 VULNERABLE PACKAGES). FOR EACH VARIANT, WE RECORD THE NUMBER OF VULNERABLE PACKAGES, EXPLOITABLE ENTRY POINTS (EXP. EP), THE NUMBER OF MODULE PATHS AND ACCESSIBLE ENTRY POINTS (AEP) DISCOVERED.

Variant	# Vuln. Pkg	# Exploitable EP
Bullseye	290	818
Bullseye$_{\text{NoVM}}$	283	711
Bullseye$_{\text{NoPW}}$	176	357
Bullseye$_{\text{ArteauSE}}$	164	333
Bullseye$_{\text{NoVM, NoPW}}$	163	335
Bullseye$_{\text{NoPW, ArteauSE}}$	157	317
Bullseye$_{\text{ArteauFuzzy}}$	147	291

TABLE V
COMPARISON OF BULLSEYE VARIANTS MUTATED IN THE GUIDED EXECUTION STAGE (FOR ALL 290 VULNERABLE PACKAGES).

V. RELATED WORK

Several comprehensive studies in recent years have focused on prototype pollution vulnerabilities. In this section, we discuss the most relevant ones to our study (see also Appendix D).

Dynamic analysis on prototype pollution. Dynamic analysis is a runtime-based approach that inspects a program's internal state during execution, thereby streamlining the detection of prototype pollution. As the analysis occurs in runtime mode, it can naturally capture and interpret runtime semantics, including those introduced by dynamic language features. These semantics are available to access and utilize using JavaScript's reflection and dynamic methods. One notable work leveraged the language's built-in features to perform dynamic analysis is the work by Arteau [4]. The work presented a lightweight dynamic analysis to detect prototype pollution vulnerabilities. This method invokes functions dynamically with a pre-defined set of inputs and monitors for signs of prototype pollution. The approach uses JavaScript's reflection capabilities to observe the program's side effects, checking whether injected properties propagate into the prototype chain. Arteau detected prototype pollution vulnerabilities in 15 packages.

Although Arteau [4] mostly avoids false positives by directly observing execution behavior, it has notable limitations. The approach employs a list of 12 exploit inputs, defined based on the signature of functions that are commonly vulnerable to prototype pollution (e.g., merge, copy, extend). Consequently, the analysis coverage is restricted to the targets that match these signatures. Also, since this technique operates as a black-box testing, it provides no insight into the location or specific code responsible for the vulnerability. Zhou and Gao [30] addressed the limited test coverage in Arteau's work by extending the exploit list to 44 (i.e., 32 new exploit inputs). This extension allowed the authors to discover 65 prototype pollution vulnerabilities (resulting in 6 CVEs). However, while this extension improved the false negative rate, the use of fixed inputs is still a significant barrier for code coverage (e.g., where the exploit needs to follow the function signature to properly execute the vulnerability).

Bullseye builds on the basic idea proposed by Arteau [4] and Zhou and Gao [30], leveraging test cases to assess whether an entry point is exploitable. However, Bullseye's novel redesign in various aspects significantly improves detection efficiency, and therefore, outperforms both of them.

Static analysis on prototype pollution. In static analysis, the source (or compiled) code of a program is examined without actually running it. Specifically, it applies abstract syntax tree (AST), control flow graph (CFG), or a combination of both, to identify patterns specifically designed for prototype pollution. Kim et al. [14] identify such common patterns and use AST and CFG to check if these patterns are found in a given package, and then use data flow analysis to track inputs from attacker-controlled sources (where data enters) to sensitive sinks (where it is consumed). From a dataset of 30,000 top-downloaded packages, their tool, DAPP identified 75 of them to be vulnerable. Through manual verification, the authors confirmed 37/75 of those vulnerabilities are true positives (resulting into 24 CVEs). DAPP is reported to be efficient—taking 0.35 seconds/package, when analyzed 100,000 packages with 25 computers (Intel i7-47903.60GHz, 16GB RAM)—but fails to analyze 25.68% of the packages. Similar to DAPP, Bullseye adopts an AST-based static analysis, but limited to the testsuites only, from which we extract the test cases corresponding to their entry points.

Kluban et al. [15] designed the framework that automatically crawled Snyk and VulnCodeDB to aggregate functions with verified prototype pollution vulnerabilities, thereby constructing a dedicated vulnerability database. Then they applied static analysis to match functions under test against the known vulnerable patterns, for both prototype pollution and Regular Expression Denial of Service (ReDoS). This framework also applied static multi-file taint analysis by tracing the dependencies between modules. It detected 290 zero-day cases across 134 packages (from 3,000), with 25 CVEs published.

ObjLupAnsys by Li et al. [16] improved the abstract interpretation by supporting object lookup analysis, in which they extended the traditional taint tracking to include the taint between objects and properties as data-flow edges. 48,162 NPM packages with over 1,000 weekly downloads were crawled and tested with ObjLupAnsys, with 61 new vulnerabilities were uncovered, leading to 11 CVEs. Li et al. [17] later consolidated a suite of tools into ODGen, with ObjLupAnsys serving as the component specifically designed for detecting prototype pollution vulnerabilities. Applying ODGen to a broader set of packages, they identified 19 instances of prototype pollution, four of which were assigned CVE identifiers. However, this approach faces scalability issues due to path explosion, a problem where the number of execution paths grows exponentially with the size and complexity of the program [11], [30]. A newer proposal called FAST [11] resolves the scaling issues in ODGen, but does not consider prototype pollution as it cannot be modeled by one taint flow (mentioned by the authors). Note that our side-effect monitoring is designed to get as close as possible to the sink location resolution typically achieved by static analysis tools like ODGen.

Prototype pollution gadget detection. In addition to prototype pollution, other studies have also focused on detecting attack chains associated with prototype pollution. Shcherbakov et al. [25] proposed an attack chain leveraging prototype pollution: first, an attacker injects malicious data into untrusted parts of the Node.js application through prototype pollution; then a code snippet (gadget), spreads the attacker-controlled data to a critical security endpoint. The authors therefore developed Silent-Spring to detect such vulnerabilities. Silent-Spring employed both static taint analysis to identify prototype pollution in Node.js packages and applications, and a hybrid approach combining dynamic and static analysis to detect the gadget that can spread the injected data from prototype pollution. With this integrated framework, the authors found 11 universal code snippet in Node.js source code, and manually exploited eight vulnerabilities in three prominent applications.

Similarly, to identify gadgets in server-side Node.js applications, which can be chained by prototype pollution, Shcherbakov et al. [26] developed Dasty that uses dynamic taint analysis. Driven by the existing testsuites in the packages under test, Dasty leverages dynamic AST-level instrumentation to identify potentially vulnerable code flows. As Dasty is sorely responsible for identifying potential gadgets, the authors integrated the toolchain of Silent-Spring, which detects prototype pollution. Ultimately, Dasty found that 631 packages with code flows that may reach dangerous sinks. Through manual analysis, the authors confirmed and built proof-of-concept exploits for 49 Node.js packages (with one CVE). Recently, Liu et al. [18] proposed the Undefined-oriented Programming Framework (UoPF), which uses concolic execution with undefined properties as symbols to detect and chain gadgets in prototype pollution attacks. This approach enables the discovery of complex gadget chains that cannot be easily captured by other tools. UoPF detected 25 zero-day gadgets, five of which were fixed after responsible disclosure.

These tools focus on discovering gadgets exploitable for prototype pollution attacks; Bullseye can complement them by identifying prototype pollution entry points and sinks, which may discover more attack chains.

Testsuite-guided detection. Many developers provide use case examples specifically designed for entry points within their packages. These testsuites offer insights for generating test cases for various purposes. To detect exploitable gadgets, Cornelissen and Shcherbakov [7] designed GHunter with customized runtimes (i.e., Node.js, Deno) and the V8 engine. Given a target runtime, GHunter drives it with its own testsuites, and performs dynamic taint analysis on it. GHunter discovered 56 new gadgets in Node.js and 67 gadgets in Deno. Similarly, Luo et al. [19] utilized testsuites to automatically generate end-to-end application inputs for vulnerability validation (covering payloads for XSS, SQLi, and prototype pollution), employing a trace-guided mutation mechanism based on concolic execution. They tested 15 Node.js web applications, and detected 20/26 known vulnerabilities. Both studies leveraged testsuites to guide their analysis—one to identify gadgets and the other to locate application-accessible functions. In contrast, our work employs testsuites as a source of suitable inputs (i.e., argument values and types) in our detection of prototype pollution vulnerabilities.

VI. Limitations

The main limitations of Bullseye include the following. *(1) Incomplete matching of entry points with corresponding function calls in testsuites:* Our current matching technique relies on pre-defined string patterns, which may lead to false negatives when path aliases, dynamic imports, or unconventional project structures are involved. This could be addressed by employing a more generalizable method, such as analyzing function signatures derived from the codebase, or leveraging static analysis techniques to more accurately attribute function calls to the correct module. *(2) Missing candidate exploit generations:* Some complex data structures, such as when test inputs are JavaScript objects with nested properties, are not handled for efficiency reasons. In those cases, both data and control might be embedded in object properties, which we do not down to create fine-grained combinations at the level of individual properties. *(3) Missing sink locations:* Bullseye does not detect all sink locations, primarily due to the reliance on passing the proxied object as an argument to an entry point. This strategy fails in cases where the target object is not explicitly passed, such as when a variable is defined directly within the function body. This limitation could be addressed through an instrumented Node.js runtime. *(4) Incomplete cross-reference with historic CVEs:* We match entry points and sink locations with GitHub advisories by comparing function names and sink paths marked with backticks (`), as commonly used in MarkdownGuide.org. However, this may miss matches when advisories are not written in Markdown, especially those imported from other platforms. This can be addressed by using commit links from advisories to match patched code lines with our detected sink locations.

VII. Conclusion

Although there is a significant body of work on prototype pollution vulnerability detection, as evident from our evaluation in terms of finding hundreds of zero-days, including in many highly-downloaded Node.js packages, more research effort is required in this area. An inherent reason is the dynamic nature of JavaScript that results into significant complexities for static analysis. We hope that insights from the design and evaluation of Bullseye will help future work in detecting/preventing prototype pollution vulnerabilities. We will make our code and evaluation artifacts available to researchers via https://github.com/Madiba-Research/Bullseye.

VIII. Ethics Considerations

Our work has obvious ethical implications, as we identify zero-day vulnerabilities in widely-used software packages, which may affect many server-side applications that use such vulnerable packages, and of course, in-turn, users of those services. We reached out to developers of each vulnerable package, with all the details of the vulnerabilities (including the exploits), either through emails or dedicated bug-reporting platforms; messages are re-sent if no response is received in 3 days. In total, we wait for four weeks before we make the vulnerability public through a GitHub advisory, and submit a CVE. One developer (for the package 'node-opcua-alarm-condition@2.134.0') requested for two additional weeks, and we waited about two months in that case before making the advisory public. Maintainers of 4 packages also awarded us bug bounties.

In terms of disclosure timeline, for the 62 CVEs (listed in the Appendix), 21 were published following an email notification, with an average of 127 days. For the remaining 41 CVEs—where no email notification was received, we used the publication date listed on `cve.org` (which may initially contain only the CVE ID): the average delay was 106 days. Only 3 CVEs were published under 60 days: 2 were fixed soon after our notification, for another, the developer fixed the issue in a new version (not the reported version). 20 CVEs were published after 79 days (in batch) for which no developer responses were received. We resort to a 30-day timeline (from an initial 90-day timeline), as we observed that CVEs take long before they are made public, and some developers fix a bug only after a CVE becomes public. We always allow the affected developers to take longer if needed, and have received no objections from developers so far.

Note that some packages are not patched when we make the vulnerability public, for various reasons: some developers take months to apply a fix (recall Fig. 3); and some do not want to fix the bugs as the vulnerable entry points are not listed in the package documentation (i.e., not expected to be used by application developers). Attackers may take advantage of these publicly listed but unpatched vulnerabilities. However, we believe that application developers using such vulnerable packages should also be aware of these vulnerabilities, so that they can find a non-vulnerable alternative—to reduce harm on their services and users.

Acknowledgment

We are grateful to the anonymous NDSS 2026 reviewers for their insightful suggestions and comments, and for guiding us in the final version of this paper. We also appreciate the help we received from the members of Concordia's Madiba Security Research Group, specifically Wanjun Gong for assisting on our CVE submissions. This work is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[1] acornjs/acorn, "Acorn AST walker," https://www.npmjs.com/package/acorn-walk.

[2] ——, "A small, fast, JavaScript-based JavaScript parser," https://github.com/acornjs/acorn.

[3] A. AlHamdan and C.-A. Staicu, "Welcome to Jurassic park: A comprehensive study of security risks in Deno and its ecosystem," in *Network and Distributed System Security (NDSS) Symposium*, San Diego, CA, USA, 2025.

[4] O. Arteau, "Prototype pollution attack in NodeJS application," 2018, North-sec conference, Montreal, Canada. https://github.com/HoLyVieR/prototype-pollution-nsec18/.

[5] M. Bentkowski, "Exploiting prototype pollution - RCE in Kibana (CVE-2019-7609)," technical report (Oct. 30, 2019). https://research.securitum.com/prototype-pollution-rce-kibana-cve-2019-7609/.

[6] D. Cassel, N. Sabino, L. Jia, and R. Martins, "NODEMEDIC-FINE: Automatic Detection and Exploit Synthesis for Node.js Vulnerabilities," in *Network and Distributed System Security Symposium (NDSS'25)*, San Diego, CA, USA, Feb. 2025.

[7] E. Cornelissen, M. Shcherbakov, and M. Balliu, "GHunter: Universal prototype pollution gadgets in JavaScript runtimes," in *Usenix Security Symposium*, Philadelphia, PA, USA, Aug. 2024.

[8] J. Czerwonka, "Pairwise testing in the real world: Practical extensions to test-case scenarios," in *Pacific Northwest Software Quality Conference (PNSQC'06)*, Portland, OR, USA, Oct. 2006.

[9] GitHub.com, "REST API endpoints for global security advisories," https://docs.github.com/en/rest/security-advisories/global-advisories?apiVersion=2022-11-28.

[10] isaacs/node-glob, "Glob functionality for Node.js," https://github.com/isaacs/node-glob.

[11] M. Kang, Y. Xu, S. Li, R. Gjomemo, J. Hou, V. N. 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)*, San Francisco, CA, USA, May 2023.

[12] Z. Kang, S. Li, and Y. Cao, "Probe the Proto: Measuring client-side prototype pollution vulnerabilities of one million real-world websites," in *Network and Distributed System Security Symposium (NDSS'22)*, San Diego, CA, USA, Apr. 2022.

[13] Z. Kang, M. Lyu, Z. Liu, J. Yu, R. Fan, S. Li, and Y. Cao, "Follow my flow: Unveiling client-side prototype pollution gadgets from one million real-world websites," in *IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA, May 2025.

[14] 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*, Feb. 2022.

[15] M. Kluban, M. Mannan, and A. Youssef, "On Detecting and Measuring Exploitable JavaScript Functions in Real-world Applications," *ACM Transactions on Privacy and Security (ACM TOPS)*, vol. 27, no. 1, pp. 1–37, Feb. 2024.

[16] S. Li, M. Kang, J. Hou, and Y. Cao, "Detecting Node.js prototype pollution vulnerabilities via object lookup analysis," in *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'21)*, Athens, Greece, Aug. 2021.

[17] ——, "Mining Node.js vulnerabilities via object dependence graph and query," in *Usenix Security Symposium*, Boston, MA, USA, Aug. 2022.

[18] Z. Liu, K. An, and Y. Cao, "Undefined-oriented Programming: Detecting and Chaining Prototype Pollution Gadgets in Node.js Template Engines for Malicious Consequences," in *IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA, May 2024.

[19] C. Luo, P. Li, W. Meng, and C. Zhang, "Test suites guided vulnerability validation for Node.js applications," in *ACM Conference on Computer and Communications Security (CCS'24)*, Salt Lake City, UT, USA, Oct. 2024.

[20] Microsoft, "Microsoft pairwise independent combinatorial tool (PICT)," https://github.com/microsoft/pict.

[21] Mozilla.org, "Error.prototype.stack - JavaScript," https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error/stack.

[22] ——, “Object.prototype.__proto__ - JavaScript,” https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/proto.

[23] ——, “Proxy - JavaScript,” https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy.

[24] NodeJS.org, “VM (executing JavaScript): Node.js v22.7.0 documentation,” https://nodejs.org/api/vm.html.

[25] M. Shcherbakov, M. Balliu, and C.-A. Staicu, “Silent Spring: Prototype pollution leads to remote code execution in Node.js,” in *Usenix Security Symposium*, Anaheim, CA, USA, Aug. 2023.

[26] M. Shcherbakov, P. Moosbrugger, and M. Balliu, “Unveiling the invisible: Detection and evaluation of prototype pollution gadgets with dynamic taint analysis,” in *The ACM Web Conference 2024 (WWW’24)*, Singapore, May 2024.

[27] K. Tatsumi, “Test case design support system,” in *International Conference on Quality Control (ICQC)*, Tokyo, Japan, Oct. 1987.

[28] VisualStudio.com, “Visual Studio Code: glob patterns reference,” https://code.visualstudio.com/docs/editor/glob-patterns.

[29] L. Wachter, J. Gremminger, C. Wressnegger, M. Payer, and F. Toffalini, “DUMPLING: Fine-grained differential JavaScript engine fuzzing,” in *Network and Distributed System Security (NDSS) Symposium*, San Diego, CA, USA, 2025.

[30] P. Zhou and Y. Gao, “Detecting prototype pollution for Node.js: Vulnerability review and new fuzzing inputs,” *Elsevier Computers & Security*, Feb. 2024.

Appendix

A. Implementation

Bullseye is implemented in JavaScript, comprising 2940 lines of code. We integrate several key components as follows. The package setup leverages JavaScript’s `import` for dynamic and asynchronous loading of the target package. To enumerate exported functions, Bullseye applies `Reflect.ownKeys`, enabling identification of all entry functions within the imported package object. Test inputs are extracted by locating relevant test files within the package. We curated a list of path patterns to cover all possible locations of test files in the Node.js packages (e.g., $/**/*\{test, spec\}*.js$), based on the observation of 100 Node.js Packages. Then we use the Node.js library Glob [10] to match these patterns in the package’s directory (Table VIII in the appendix). Test files are parsed into ASTs using Acorn [2], and traversed with Acorn-walk [1] to extract the relevant test inputs for each entry point.

We rely on Microsoft PICT [20] to effectively produce exploit input candidates. For each test case, we create a PICT module with the test inputs and seeds. The PICT module combines them into a series of exploitable inputs. A PICT module consists of a set of objects, where in each object we define the property and its possible values. Thus, we assign the input’s arguments from the test input and the selected seeds from Table I (i.e., by matching the type) as the “values” of objects. Meanwhile, we generate the labels that contain the order and data type of the input arguments, and set these labels as the “properties” of the objects. This format allows reassembling the results from PICT into usable inputs. For instance, for the input $(“test”, \{\}, true)$, we generate the properties: string1, object2, boolean3, respectively. Given the first property, its corresponding value comes from the test input “test”, along with the type-matched string seeds. The final object for the first property becomes as: { `property: 'string1', values: [ 'test', '__proto__' ]` }.

We use Proxy [23] to intercept the modification attempt on Object.prototype. Since the global prototype is immutable, we wrap a Proxy on the prototype of an empty object, then we modify the `set` handler in the Proxy to detect if the function we execute attempts to modify the prototype. If the modification attempt is detected, we log the stack trace using JavaScript’s error stack (`Error.prototype.stack` [21]), which includes the sink location (i.e., last executed line after the modification attempt).

Furthermore, we use JavaScript VM [24] to mitigate infinite execution loops and unexpected behaviors that might affect the runtime during the execution. We specifically use `runInContext` to test the entry point function with a timeout threshold of 100 milliseconds. The target functions are dynamically invoked using `Reflect.apply`.

To ensure the integrity of the runtime of each package, we run Bullseye in containers. These containers are dynamically created for each package in the loop. Instead of using Docker’s CLI, we use the ‘dockerode’ library to communicate with the Docker engine through the Docker API. This approach also improves the efficiency of Bullseye. For each package under test, our system automatically creates a container based on an image we prepared with Node.js and dependent libraries for Bullseye. This container takes the package information as an input and run Bullseye. The container then listens for the detection results from Bullseye and records them in a log file. At the end of the execution (or if a timeout is reached), the container is removed, freeing up the host’s resources.

We use the package ‘p-limit’[10] to control parallel executions between containers. Specifically, we wrap each container execution in a `limit` call, ensuring that only a specified number of containers, defined as an argument, run at the same time. It also prevents system overload from too many parallel processes. This concurrency configuration allows us to manage resource usage effectively while still taking advantage of parallelism to process multiple packages simultaneously.

B. Manual Analysis of FNs in ODGen and Silent-Spring

We manually checked all 12 FNs from ODGen, and identified the following reasons for missing them in Bullseye. *(1) Infeasible attack vectors*: we identified 8 sink locations in 5 packages that need pre-conditions which are not aligned with the package’s use cases. They are: ‘class-transformer@0.2.3’, ‘dnspod-client@0.1.3’, ‘draft@0.2.3’, ‘field@1.0.1’, ‘node-file-cache@1.0.2’. For instance, in the package ‘class-transformer@0.2.3’, the object ‘payload’ is modified at its prototype with a new property, which equals the payload itself (see line 3 in Listing 3, in the appendix); also the ‘toString()’ property in the payload is assigned with an anonymous function that returns the polluted value. *(2) Complex exploits*: we identified 4 sink locations where Bullseye failed because of multi-step exploits (in packages ‘bayrell-nodejs@0.8.0’ and ‘grunt-util-property@0.0.2’). For example,

[10] https://www.npmjs.com/package/p-limit

to exploit 'grunt-util-property@0.0.2', one need to call the method `addExport` as follows (`use.addExport({}, {getClassName: function() { return "FRAG#15"; }, toString: function() { return "VALUE"; } });`), where the function `getClassName` return the payload string that matches the fragment #15 in Table I. Bullseye missed the vulnerability as the exploit is not in the pre-defined fixed inputs, and the package has no testsuites to help us generate the exploit.

We identified the following reasons for the 40 FNs from Silent-Spring: complex exploits (23), infeasible attack vector (7), unknown payload pattern (8), and 4 false positives (1 in 'deephas@1.0.5', and 3 in 'dot-object@2.1.2'). As an example of the complex exploits, in 'immer@8.0.0', the exploit need to call a function 'enablePatches' before running the entry point with the payload; this multi-step exploit cannot be executed by Bullseye. Similar to ODGen, we also noticed that 7 reported vulnerabilities require initializing a property in the global prototype chain, which we cannot find in any relevant test cases. Arguably, such cases should not be considered true positives as the prerequisite conditions do not align with a package's use cases. Eight vulnerabilities Bullseye failed to detect because of unknown fragments (i.e., not in our Table I). For instance, the package 'arr-flatten-unflatten@1.1.4' is exploitable through the exploit `unflatten({ "__proto__[polluted]": "yes"});`.

C. Prototype Pollution Vulnerability

The flexibility of JavaScript enables developers to create dynamic features of an object with ease, but this same flexibility can introduce security challenges when objects are not adequately protected. Prototype pollution arises from unsafe manipulation of JavaScript's prototype chain, exploiting its dynamic object model. This vulnerability allows attackers to inject or modify properties that can affect all objects inheriting from the compromised prototype, leading to privilege escalation, denial of service, command execution, etc. More concretely, consider the following code fragment: $victim[prop1][prop2] = value$; the vulnerability occurs when an attacker can control at least the first property 'prop1' and the assigned 'value', the two properties 'prop1' and 'prop2', or all of the three identifiers. In all these cases, the attacker should control 'prop1' to supply the keyword __proto__ (a built-in prototype setter), which makes the object 'victim' to expose the value of its prototype [22]. Then, for the first case (i.e., 'prop1' and 'value' are controllable), the attacker can alter the value of 'prop2' with an injected value (e.g., 'isAdmin'), affecting any object in the program that uses 'prop2' in its logic. The attacker can set ('__proto__', 'true'), which creates the construct: $victim[\text{``}\_\_proto\_\_\text{''}][\text{``}isAdmin\text{''}] = true$, causing all objects in the program to get the property 'isAdmin' with 'true', possibly leading to privilege escalation. The other case is when the two properties 'prop1' and 'prop2' are attacker-controllable. In this case, two types of attack can be launched. First is denial-of-service, in which the attacker can modify an existing property or method (e.g., 'toString', 'valueOf'), by supplying the name of this method to 'prop2' and assign an arbitrary value, which results in the construct: $victim[\text{``}\_\_proto\_\_\text{''}][\text{``}toString\text{''}] = 123$, potentially rendering some part of the program unavailable. The second attack is Arbitrary Command Execution (ACE), in which the attacker uses 'prop2' to pass a special property name called universal gadget [25], [18], [7]. Such gadgets can be used in a code execution sink, such as `exec`, allowing the attacker to inject an arbitrary command to be executed by the sink [5].

D. Other Related Studies

Kang et al. [12] use dynamic taint analysis to detect client-side prototype pollution in websites. Later work [13] improves this approach by guiding injected properties into sinks using values from non-vulnerable websites, and identified 133 new gadgets, resulting in a CVE. Cassel et al. [6] combine dynamic taint tracking with type-aware and structure-aware fuzzing to improve exploit generation for ACE (arbitrary code execution) and ACI (arbitrary command injection) vulnerabilities in NPM packages. Their work also resulted in the assignment of a high-severity CVE. Watcher et al. [29] propose DUMPLING, a differential fuzzer for JavaScript engines that detects JIT compilation vulnerabilities by comparing fine-grained execution states between optimized and normal code paths. This method discovered eight new bugs in the V8 engine. AlHamdan et al. [3] evaluated the security features of the new JavaScript runtime Deno, and showed that despite its stricter permission model, it still suffers from known issues such as ReDoS, prototype pollution, and permission misuse in its ecosystem. Bullseye can improve this work for finding more prototype pollution vulnerabilities, e.g., by integrating our enhanced side-effect checking oracles.

E. Other Code Listing and Tables

```javascript
const root = require("./class-transformer@0.2.3");
const payload = JSON.parse('{"__proto__": {"polluted": "yes"}}');
payload.__proto__.polluted = payload;
payload.toString = function () {
  return "yes";
};
root.classToClassFromExist(payload, {}, { enableCircularCheck: true });
```

Listing 3. PoC exploit for class-transformer@0.2.3

Package	CVSS (Severity)	CVE ID
fast-loops@1.1.3	10 (CRITICAL)	CVE-2024-39008
ag-grid-community@31.3.2	9.8 (CRITICAL)	CVE-2024-38996
ag-grid-enterprise@31.3.2	9.8 (CRITICAL)	CVE-2024-38996
@ag-grid-enterprise/charts@31.3.2	9.8 (CRITICAL)	CVE-2024-38996
@agreejs/shared@0.0.1	9.8 (CRITICAL)	CVE-2024-39017
@cafebazaar/hod@0.4.14	9.8 (CRITICAL)	CVE-2024-39015
@blackprint/engine@0.9.1	9.8 (CRITICAL)	CVE-2024-24294
getsetprop@1.1.0	9.8 (CRITICAL)	CVE-2024-36575
@jsonic/jsonic-next@2.12.1	9.8 (CRITICAL)	CVE-2024-38993
@almela/obx@0.0.4	9.8 (CRITICAL)	CVE-2024-36573
@chargeover/redoc@2.0.9-rc.69	9.8 (CRITICAL)	CVE-2024-39011
@allpro/form-manager@0.7.4	9.8 (CRITICAL)	CVE-2024-36572
mini-deep-assign@0.0.8	9.8 (CRITICAL)	CVE-2024-38983
@thi.ng/paths@5.1.62	9.8 (CRITICAL)	CVE-2024-29650
@chasemoskal/snapstate@0.0.9	9.8 (CRITICAL)	CVE-2024-39010
@75lb/deep-merge@1.1.1	9.8 (CRITICAL)	CVE-2024-38986
json-override@0.2.0	9.8 (CRITICAL)	CVE-2024-38984
@cdr0/sg@1.0.10	9.8 (CRITICAL)	CVE-2024-36580
2o3t-utility@0.1.2	9.8 (CRITICAL)	CVE-2024-39013
@cahil/utils@2.3.2	9.8 (CRITICAL)	CVE-2024-39014
@ais-ltd/strategyen@0.4.0	9.8 (CRITICAL)	CVE-2024-39012
@bunt/util@0.29.19	9.8 (CRITICAL)	CVE-2024-38989
@andrei-tatar/nora-firebase-common@1.12.2	9.8 (CRITICAL)	CVE-2024-30564
@alexbinary/object-deep-assign@1.0.11	9.8 (CRITICAL)	CVE-2024-36582
chartist@1.3.0	9.8 (CRITICAL)	CVE-2024-45435
utils-extend@1.0.8	9.1 (CRITICAL)	CVE-2024-57077
@intlify/message-resolver@9.1.10	8.9 (HIGH)	CVE-2025-27597
@airvertco/frappejs@0.0.11	8.8 (HIGH)	CVE-2024-38992
@akbr/patch-into@1.0.1	8.8 (HIGH)	CVE-2024-38991
@bit/loader@10.0.3	8.8 (HIGH)	CVE-2024-24293
requirejs@2.3.6	8.4 (HIGH)	CVE-2024-38998
@apphp/object-resolver@3.1.1	8.3 (HIGH)	CVE-2024-36577
uplot@1.6.30	8.2 (HIGH)	CVE-2024-21489
dset@3.1.3	8.2 (HIGH)	CVE-2024-21529
@apidevtools/json-schema-ref-parser@11.1.0	8.1 (HIGH)	CVE-2024-29651
@c3/utils-1@1.0.131	8.1 (HIGH)	CVE-2024-39016
@byondreal/accessor@1.0.0	8.1 (HIGH)	CVE-2024-36583
@abw/badger-database@1.2.1	7.6 (HIGH)	CVE-2024-36581
web3-utils@4.2.0	7.5 (HIGH)	CVE-2024-21505
@amoy/common@1.0.10	7.3 (HIGH)	CVE-2024-38994
@stryker-mutator/util@8.2.6	7.3 (HIGH)	CVE-2024-57085
dot-properties@1.0.1	7.5 (HIGH)	CVE-2024-57084
@zag-js/core@0.49.0	7.5 (HIGH)	CVE-2024-57079
underscore-contrib@0.3.0	7.5 (HIGH)	CVE-2024-57081
xe-utils@3.5.26	7.5 (HIGH)	CVE-2024-57074
vxe-table@4.8.10	7.5 (HIGH)	CVE-2024-57080
ajax-request@1.2.3	7.5 (HIGH)	CVE-2024-57076
eazy-logger@4.0.1	7.5 (HIGH)	CVE-2024-57075
node-opcua-alarm-condition@2.124.0	7.5 (HIGH)	CVE-2024-57086
cli-util@1.1.27	7.5 (HIGH)	CVE-2024-57078
module-from-string@3.3.1	7.5 (HIGH)	CVE-2024-57072
@ndhoule/defaults@2.0.1	7.5 (HIGH)	CVE-2024-57066
@syncfusion/ej2-spreadsheet@25.2.4	7.5 (HIGH)	CVE-2024-57064
utile@0.3.0	7.5 (HIGH)	CVE-2024-57065
php-parser@3.1.5	7.5 (HIGH)	CVE-2024-57071
expand-object@0.4.2	7.5 (HIGH)	CVE-2024-57069
php-date-formatter@1.3.6	7.5 (HIGH)	CVE-2024-57063
dot-qs@0.2.0	7.5 (HIGH)	CVE-2024-57067
@tanstack/form-core@0.19.5	7.5 (HIGH)	CVE-2024-57068
@stryker-mutator/util@8.2.6	7.5 (HIGH)	CVE-2024-57085
redoc@2.2.0	7.5 (HIGH)	CVE-2024-57083

TABLE VI
The list CVEs from our work (critical and high severity). Note that in CVE-2024-38996, four vulnerable packages from the same vendor are grouped.

No.	Path Patterns	No.	Path Patterns
1	PACKAGE	14	./
2	./PACKAGE	15	..
3	../PACKAGE	16	../../
4	../../PACKAGE	17	./index.js
5	../	18	../index.js
6	./src/**	19	./lib/**
7	../src/**	20	../lib/**
8	../../src/**	21	../../lib/**
9	../src/index	22	../lib/index
10	../../src/index	23	./src/PACKAGE
11	../src/PACKAGE	24	../../src/PACKAGE
12	./lib/PACKAGE	25	../lib/PACKAGE
13	../../lib/PACKAGE		

TABLE VII
List of path patterns used for locating package imports

<table>
<tr><th>No.</th><th>Patterns</th></tr>
<tr><td>1</td><td>fnDir || ** || fnName/fnName || *{,.,-}preSuf.{js,coffee,ts,cjs,mjs}</td></tr>
<tr><td>2</td><td>fnDir || ** || fnName/preSuf{,.,-}fnName || *.{js,coffee,ts,cjs,mjs}</td></tr>
<tr><td>3</td><td>fnDir || ** || fnName/packageName || *{,.,-}preSuf.{js,coffee,ts,cjs,mjs}</td></tr>
<tr><td>4</td><td>fnDir || ** || fnName/preSuf{,.,-}packageName || *.{js,coffee,ts,cjs,mjs}</td></tr>
<tr><td>5</td><td>**/packageName || *{,.,-}preSuf.{js,coffee,ts,cjs,mjs}</td></tr>
<tr><td>6</td><td>**/preSuf{,.,-}packageName || *.{js,coffee,ts,cjs,mjs}</td></tr>
<tr><td>7</td><td>{test,Test,__tests__,__Tests__,tests,Tests,spec,Spec,coffee,Coffee}/**/*.{js,cjs,mjs}</td></tr>
<tr><td>8</td><td>*{Test,test,Spec,spec}*.{js,cjs,mjs}</td></tr>
<tr><td>9</td><td>{test,Test,__tests__,__Tests__,tests,Tests,spec,Spec}/*.{js,cjs,mjs}</td></tr>
<tr><td>10</td><td>*{Test,test,Spec,spec}*.{js,cjs,mjs}</td></tr>
</table>

TABLE VIII
List of glob patterns used for locating testsuite files in Node.js packages. Notation: 'fnDir': the function path created from the given function name, covering cases where the test files hierarchy are derived from the function's path (e.g., assign/object/merge.js is converted from assign.object.merge); 'fnName': the function name without the path (the last string in a dot-separated name), to cover cases where the test file is named after the last name in the function path (e.g., merge.js is converted from util.merge); 'packageName': the name of the package; 'preSuf': an array with the values: [{t,T}est, {s,S}pec, {i,I}ndex, {c,C}offee].

No.	Exploit Inputs	No.	Exploit Inputs
1	BAD_JSON	23	“this.constructor.prototype.test”, {}, “123”
2	BAD_JSON, {}	24	“__proto__.test”, “123”, {}
3	{}, BAD_JSON	25	{}, “/__proto__/test”, “123”
4	BAD_JSON, BAD_JSON	26	{}, “/__proto__/test”, “123”, true
5	{}, {}, BAD_JSON	27	“__proto__.test=123”
6	{}, {}, {}, BAD_JSON	28	“__proto__:test”, “123”
7	{}, “__proto__.test”, 123	29	“__proto__[test]=123”, {}
8	{}, “__proto__[test]”, 123	30	{}, “constructor/prototype/test”, “123”, “/”
9	“__proto__.test”, 123	31	“__proto__”, { “test”: “123” }, {}, true
10	“__proto__[test]”, 123	32	{ “__proto__.test”: “123” }
11	{}, “__proto__”, “test”, 123	33	{ “constructor.prototype.test”: “123” }
12	“__proto__”, “test”, 123	34	{}, [[“__proto__”], “test”], “123”
13	{}, BAD_JSON, {}	35	[[“__proto__”], “test”], “123”, {}
14	{}, BAD_JSON, true	36	{}, [[“__proto__”], “test”], “123”, true
15	true, {}, BAD_JSON	37	{}, [“__proto__”, “test”], “123”
16	{}, true, BAD_JSON	38	{}, [“constructor.prototype.test”], “123”
17	true, {}, BAD_JSON2	39	[“__proto__”], “test”, “123”
18	{}, BAD_JSON2	40	[“__proto__.test”], [“123”]
19	BAD_JSON2	41	{}, [[“__proto__”], [“__proto__”], “test”], “123”
20	“[__proto__]\ntest=123”	42	[“-constructor.prototype.test”, “123”]
21	{}, “constructor.prototype.test”, “123”	43	“filename” -> [__proto__]\ntest=“123”
22	“__proto__.test”, {}, “123”	44	“filename” -> [constructor]\nprototype.test=“123”

TABLE IX
List of exploit inputs curated by Arteau [4] (the first 12), and Zhou and Gao [30]. Notation: (\n, ->) refer to a new line-separated payload, and file's content payload, respectively; (BAD_JSON, BAD_JSON2) refer to the following JSON-based payloads, respectively: JSON.parse('{"__proto__":{"test":123}}') and JSON.parse('{"constructor":{"prototype":{"test":123}}}').