Verification and Classification of Exploits for Node.js Vulnerabilities

Sungmin Park
Korea University
ryan040@korea.ac.kr

***Abstract*—Vulnerabilities in the Node.js ecosystem pose serious security threats. Generating exploits for such vulnerabilities is a critical and essential step for fixing the vulnerabilities and understanding attack vectors. To address this need, prior work has proposed a range of methods, including static analysis approaches, dynamic analysis approaches, and LLM-based techniques. However, most studies verify only at the end of execution whether the expected effect of each vulnerability has occurred. This approach does not confirm whether the exploit actually reaches the target vulnerable sinks. As a result, it may fail to exercise the intended vulnerability or inadvertently trigger a different sink.**

In this study, we propose a method for validating and classifying exploits related to Node.js vulnerabilities. Our method instruments sink APIs and related objects prior to execution to capture sink APIs calls and their arguments when a sink is triggered at runtime. This lets us verify that an exploit reaches the intended sink and classify exploits by the point at which the sink is triggered.

I. Introduction

JavaScript is used across browsers, servers, desktop, and mobile environments, and Node.js has become a general-purpose runtime spanning these domains. The npm registry that underpins this ecosystem hosts more than three million packages, which accelerates development but also greatly expands the attack surface. In npm's dense dependency graphs [1], transitive vulnerabilities mean that a flaw in a single dependency can ripple outward and expose thousands of downstream applications.

Generating exploits is essential for both remediation and understanding real attack vectors. Earlier work SecBench.js [2] was based on a corpus of exploits that had been manually crafted and curated. However, the growing volume of disclosed vulnerabilities has driven automation, FAST [3] proposed a static analysis framework, NodeMedic [4, 5] applied dynamic analysis, Explode.js [6] employed symbolic execution, and most recently PoCGen [7] introduced an LLM-based exploit generation approach. These approaches target some or all of the five vulnerability categories prototype pollution, command injection, code injection, regular expression denial of service (ReDoS), and path traversal. They have contributed significantly to advancing exploit generation. However, most of these approaches verify only whether the effects associated with each vulnerability occur at the end of execution. As shown in Fig. 1, SecBench.js [2] evaluates success using an end-state oracle. For the `sahmat` library case, an exploit is deemed

Fig. 1: Simplified Mislabeled Prototype Pollution Exploit

```
console.log({}.polluted); // "undefined"

const pkg = require("sahmat");
let obj = { tmp: "" };
pkg(obj, "", obj.__proto__.polluted = "yes");

console.log({}.polluted); // "yes"
```

successful when, at the end of execution, an unmodified object exposes the injected property `polluted`.

In this example, however, the pollution occurs during argument evaluation rather than via the intended sink, the third argument `obj.__proto__.polluted = "yes"` pollutes the global object prototype before the function containing the sink even executes. Therefore, the presence of `{}.polluted` at the end, which is only a check of a property on an unmodified object, is insufficient to demonstrate that the sink under test caused the observed effect, and hence cannot by itself justify the mislabeling. Because this procedure inspects only the end-state, the observed effect may arise from a different sink or from side effects that bypass the intended sink altogether. This risk is amplified with LLM-generated exploits hallucinations may cause the exploit itself to produce the desired effect, and although such cases can be filtered post hoc, complete elimination is difficult in practice.

To address these limitations, we instrument sink APIs and related objects with lightweight runtime hooks that intercept invocations and capture the called API and its arguments when a sink is triggered. Using this event-level evidence, we verify sink reachability and classify each exploit by the concrete trigger point rather than inferring from end-state effects, reducing false positives from look-alike effects that do not trigger the intended sink.

II. Approach

In this section, we present our method for validating and classifying exploits targeting Node.js vulnerabilities. We aggregate exploits from existing tools and normalize them to be executed in a Node.js environment without any other libraries. For each of the four vulnerability categories (prototype pollution, command injection, code injection, and ReDoS), we instrument lightweight runtime hooks prior to execution, without incurring parsing overhead, to bind each exploit to a concrete sink.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00390

TABLE I: Vulnerability categories and sink APIs

Category	Sink APIs
Command injection	`child_process.{exec[Sync], execFile[Sync], spawn[Sync], fork}`
Code injection	`global.eval`; `Function` (constructor); `vm.{runIn*, compileFunction, Script}`
Prototype pollution	`Object.{defineProperty, defineProperties}`
ReDoS	`RegExp.{exec, test, compile}`; `String.{match, replace, split, matchAll}`

Below we detail the instrumentation strategy, first distinguishing API-centric and object-centric sinks, then describing per-category instantiations and the resulting validation/classification logic.

a) API-centric sinks: We wrap selected sink APIs prior to execution so that upon invocation the wrapper logs the function identifier, arguments, and call-site location. In addition, we extract attacker-controlled key candidates from the exploit and check whether these tokens appear in the arguments observed at the sink call site. This lightweight content matching links the observed invocation to the exploit payload, providing precise, low-overhead evidence that an exploit actually triggered the intended sink. As shown in Table I, we identify four categories of sinks.

- *Command injection*: arises when attacker-controlled input is forwarded to `child_process.*` (e.g., `exec`, `spawn`), enabling unintended shell execution.
- *Code injection*: occurs when attacker-controlled code is evaluated via `eval`, the `Function` constructor, or `vm.*` (including string-based timers), thereby executing injected code.
- *Prototype pollution*: results from writing attacker-controlled keys to `Object.prototype`, for example via `Object.{defineProperty, defineProperties}`, which then surface as unexpected properties on objects.
- *ReDoS*: stems from inefficient regular expressions applied to attacker-controlled input, causing catastrophic backtracking and CPU exhaustion.

b) Object-centric sinks: For prototype pollution, relying solely on `Object.defineProperty` is insufficient as dynamic writes to `Object.prototype` may invoke only a setter and thus evade detection. Importantly, replacing `Object.prototype` itself is not permitted by the ECMAScript specification accordingly, we augment specific properties rather than overwriting the prototype object. To address this, we extract candidate property names from the exploit and pre-install setters on `Object.prototype` for those keys, making subsequent writes observable. This instrumentation slightly affects the behavior of `hasOwnProperty` for the affected keys (they become inherited accessor properties), but the impact on verification was minimal a principled treatment is left to future work.

c) Validation and classification.: We retrieve the file and line at which a sink is triggered. Based on this evidence, we classify outcomes into four categories:

(i) **In-library sink**: a hook event is observed at a sink within the *target library*.
(ii) **Third-party sink**: a hook event is observed in a dependency (e.g., call-site paths under `node_modules/`).
(iii) **Self-effect**: a hook event is observed in the exploit itself (e.g., see Fig. 1).
(iv) **No effect**: no hook event is observed.

For cases (i) and (ii), we further classify exploits by call-site `path:line` which exploits by the precise point of sink occurrence and enables consistent labeling across tools.

III. Case Study

In this section, we present a case study of `hoek` v5.0.0, which contains a prototype-pollution vulnerability CVE-2018-3728, to demonstrate our classification technique. CVE reports typically identify sinks at the level of entry-point functions, for example by stating that `merge` and `applyToDefaults` in `hoek` are vulnerable. However, multiple such entry points can map to the same concrete sink (a specific file and line); in CVE-2018-3728 both `merge` and `applyToDefaults` lead to the sink at line 137 of `hoek/lib/index.js`.

Fig. 2: Exploit from SecBench.js

```
const pkg = require("hoek");
obj = {};
let malicious_payload =
  '{"__proto__":{"polluted":"yes"}}';
pkg.merge({}, JSON.parse(malicious_payload));
```

Fig. 3: Exploit from Explode.js

```
const pkg = require("hoek/merge");
var dst_obj = {};
var src_obj =
  { ["__proto__"]: { polluted: "yes" } };
pkg(dst_obj, src_obj);
```

Fig. 4: Simplified Exploit from PoCGen

```
const pkg = require("hoek");
const opts =
  { __proto__: { polluted: "yes" } };
const keys = ["__proto__.polluted"];
pkg.applyToDefaultsWithShallow({}, opts, keys);
```

Figs. 2, 3, and 4 show exploits generated by SecBench.js, Explode.js, and PoCGen, respectively. Although all three were intended to target the same CVE, our classification shows they actually reach different sinks in `hoek/lib/index.js`: line 137 for SecBench.js and Explode.js, and line 221 for PoCGen. Notably, this means PoCGen exercises a *different vulnerability* from the one described in the CVE report (CVE-2018-3728 at line 137). As this case illustrates, our technique classifies exploits by the exact sink that an exploit reaches, even when the end state appears the same. While this enables more precise vulnerability analysis, a limitation remains. Many vulnerabilities lack assigned CVE numbers or detailed reports that specify precise sink locations, and when such information is absent our technique cannot conclusively link a PoC to the reported vulnerability. Addressing this limitation is left to future work.

Acknowledgements

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No.RS-2024-00344597) and the Institute of Information & Communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (No.RS-2024-00440780, Development of Automated SBOM and VEX Verification Technologies for Securing Software Supply Chains)

References

[1] N. Zahan, T. Zimmermann, P. Godefroid, B. Murphy, C. Maddila, and L. Williams, "What are weak links in the npm supply chain?" in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 331–340.

[2] M. H. M. Bhuiyan, A. S. Parthasarathy, N. Vasilakis, M. Pradel, and C.-A. Staicu, "SecBench.js: An Executable Security Benchmark Suite for Server-Side JavaScript," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 1059–1070.

[3] 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)*, 2023, pp. 1059–1076.

[4] D. Cassel, W. T. Wong, and L. Jia, "NodeMedic: End-to-End Analysis of Node.js Vulnerabilities with Provenance Graphs," in *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*, 2023, pp. 1101–1127.

[5] D. Cassel, N. Sabino, M. Hsu, R. Martins, and L. Jia, "NodeMedic-FINE: Automatic Detection and Exploit Synthesis for Node.js Vulnerabilities," in *32nd Annual Network and Distributed System Security Symposium, NDSS 2025, San Diego, California, USA, February 24-28, 2025*. The Internet Society, 2025.

[6] F. Marques, M. Ferreira, A. Nascimento, M. E. Coimbra, N. Santos, L. Jia, and J. Fragoso Santos, "Automated Exploit Generation for Node.js Packages," *Proc. ACM Program. Lang.*, vol. 9, no. PLDI, Jun. 2025. [Online]. Available: https://doi.org/10.1145/3729304

[7] D. Simsek, A. Eghbali, and M. Pradel, "PoCGen: Generating Proof-of-Concept Exploits for Vulnerabilities in Npm Packages," *arXiv preprint arXiv:2506.04962*, 2025.