7-2023

NodeMedic: End-to-end analysis of Node.js vulnerabilities with provenance graphs

Darion CASSEL

Wai Tuck WONG
Singapore Management University, wt.wong.2022@phdcs.smu.edu.sg

Limin JIA

Citation

CASSEL, Darion; WONG, Wai Tuck; and JIA, Limin. NodeMedic: End-to-end analysis of Node.js vulnerabilities with provenance graphs. (2023). *2023 8th IEEE European Symposium on Security and Privacy, Euro S and P: Delft, July 3-7: Proceedings*. 1101-1127.
Available at: https://ink.library.smu.edu.sg/sis_research/8094

NodeMedic: End-to-End Analysis of Node.js Vulnerabilities with Provenance Graphs

Darion Cassel
Carnegie Mellon University
Pittsburgh, USA
darioncassel@cmu.edu

Wai Tuck Wong
Singapore Management University
Bras Basah, Singapore
wt.wong.2020@msc.smu.edu.sg

Limin Jia
Carnegie Mellon University
Pittsburgh, USA
liminjia@cmu.edu

***Abstract*—Packages in the Node.js ecosystem often suffer from serious vulnerabilities such as arbitrary command injection and code execution. Existing taint analysis tools fall short in providing an end-to-end infrastructure for automatically detecting and triaging these vulnerabilities.**

We develop NodeMedic, an end-to-end analysis infrastructure that automates test driver creation, performs precise yet scalable dynamic taint propagation via algorithmically tuned propagation policies, and exposes taint provenance information as a *provenance graph*. Using provenance graphs we develop two post-detection analyses: automated constraint-based exploit synthesis to confirm vulnerabilities; Attack-defense-tree–based rating of flow exploitability.

We demonstrate the effectiveness of NodeMedic through a large-scale evaluation of 10,000 Node.js packages. Our evaluation uncovers 155 vulnerabilities, of which 152 are previously undisclosed, and 108 were confirmed with automatically synthesized exploits. We have open-sourced NodeMedic and a suite of 589 taint precision unit tests.

1. Introduction

JavaScript has been widely used on server, desktop, and IoT platforms. The vehicle for JavaScript deployment on these platforms is the Node.js runtime, which ranks as the most widely used web framework in Stack Overflow's developer survey [75] and features a diverse ecosystem of more than 1 million packages. The widespread deployment of Node.js makes it attractive to attackers. Recent years have seen a surge in attacks that target Node.js packages [65]. Several studies have measured the security of the Node.js ecosystem and found that it is rife with packages that have vulnerabilities [33], [92], [105]. The Node.js runtime lacks mechanisms for sandboxing or moderating third-party packages and thus a single vulnerability in a package's dependencies can compromise the security of the entire package [22].

Two vulnerabilities of particular interest to this paper are arbitrary code execution (ACE) and arbitrary command injection (ACI) [17], [18]. These vulnerabilities allow attackers to run arbitrary code and commands and access the underlying operating system.

One effective method of detecting such vulnerabilities is dynamic taint analysis [81]. This technique tracks the flow of attacker-controllable data during program execution and has been applied to client-side JavaScript programs to identify cross-site scripting vulnerabilities [50], [61], [76], [77]. Critically, these tools integrate the taint analysis into a comprehensive infrastructure for automatically detecting and confirming vulnerabilities.

Dynamic taint analysis has also been applied to server-side Node.js packages [32], [41], but, compared to client-side tools, server-side tools lack end-to-end infrastructure. Existing work relies on manually-crafted test drivers to run the analysis, and requires manual triage and confirmation of vulnerabilities based on reported potentially vulnerable flows. Not coincidentally, existing work has been evaluated on a small number of packages (e.g., 22 and 21 packages for Ichnaea [41] and Affogato [32] respectively).

Desirable properties of an end-to-end infrastructure. To be practically useful, a taint analysis should be integrated into an end-to-end analysis infrastructure that has the following capabilities:

(P1): Automatic package driver generation. Unlike web applications, Node.js packages need drivers to call the exported APIs for dynamic analysis. Prior analyses require manual construction of test drivers [32], [41], [67], [91], making analyzing large sets of packages labor intensive.

(P2): Precise analysis of JavaScript. Precisely tracking taint is necessary to limit false positives, an important consideration for Node.js vulnerability detection [92]. However, prior work shows precise tainting of primitives values and built-in functions is challenging [41].

(P3): Scalable analysis that supports packages with many dependencies. Since the average Node.js package has 79 dependencies [105], scalability is critical for a tool to be useful for analyzing packages in the wild. Scaling analysis to large dependency sets is also challenging [67].

(P4): Automated confirmation of exploitable flows. To eliminate false positives, potential vulnerabilities reported by the taint analysis need to be confirmed to be exploitable. Manual confirmation is time consuming, which can cause exploitable flows to be ignored. Automating the confirmation process can significantly reduce the burden of analysts. Unfortunately, prior tools [32], [41], [67], [79], [91] all require manual confirmation.

(P5): Triage of tainted flows. When automatic confirmation fails, a rating indicating how exploitable a potential vulnerability is can help analysts triage and prioritize reported tainted flows for manual examination. Researchers have proposed approaches to quantify exploitability and help prioritize review in other domains (x86 binaries [66], Java bytecode [56]). However, existing Node.js analyses [32], [41], [67], [79] give minimal feedback: tainted data *reached* a sink or not, providing little to aid triaging.

Our goal is to implement an end-to-end analysis infrastructure for Node.js with these 5 properties in mind.

End-to-end analysis. We develop NodeMedic, a dynamic taint *provenance* tracking infrastructure for end-to-end analysis of ACE and ACI vulnerabilities in Node.js packages. (We abbreviate “taint provenance” to provenance.) NodeMedic takes as input an npm package name and version, installs the package, and to address (P1), generates a driver program to run our analysis (Section 4.1).

NodeMedic leverages source-to-source rewriting to instrument packages with mechanisms that track provenance within the package and its dependencies. To address (P2), it implements propagation policies that allow precise provenance analysis (Section 3.2.1). To address (P3), NodeMedic provides an algorithm that adjusts propagation precision (Section 3.3), allowing it to scale to packages with hundreds of dependencies.

As output, the provenance analysis produces a *provenance graph*, a data structure that stores a history of all of the operations a tainted value passed through. Provenance graphs’ comprehensiveness makes them suitable to serve as the foundation of diverse post-detection analyses. To address (P4), we implement a novel constraint-based synthesis algorithm to produce candidate exploits from provenance graphs (Section 4.2). To address (P5), we connect provenance graphs to Attack-defense trees [45] to estimate flow exploitability (Section 4.3).

We evaluate NodeMedic on a set of 10,000 Node.js packages (Section 5.5) and uncover 155 vulnerabilities, of which 152 are *previously undisclosed*, and 108 are *automatically confirmed* by our exploit synthesis methodology.

Contributions. In summary, our key contributions are:

- End-to-end infrastructure design for dynamic taint analysis of Node.js packages.
- Precise taint provenance analysis producing provenance graphs that can aid vulnerability triage.
- Scalable analysis of large numbers of dependencies via automatically tuned propagation precision.
- Constraint-based automated exploit synthesis and confirmation using provenance graphs.
- An Attack-defense-tree–based approach to quantify flow exploitability using provenance graphs.
- Discovery of 152 new vulnerabilities.

We have open-sourced NodeMedic along with our suite of 589 taint precision unit tests: https://github.com/NodeMedicAnalysis/.

Responsible disclosure. We are in the process of contacting package maintainers and reporting vulnerabilities. We detail our disclosure process in Appendix D.3.

2. Threat Model and Overview

We describe the scope of NodeMedic’s threat model in Section 2.1. Then, in Section 2.2, we provide an overview of NodeMedic’s provenance analysis and describe the components of its end-to-end infrastructure.

2.1. NodeMedic Threat Model

Node.js is a JavaScript runtime built on top of the V8 JavaScript engine. Node.js developers combine code into *packages*, which can import other packages as *dependencies* to use their public APIs (exported functions). Node.js provides powerful *sensitive APIs* [63], [64], [69], [71] that can dynamically generate code and access operating system functionality (e.g., process creation, file management).

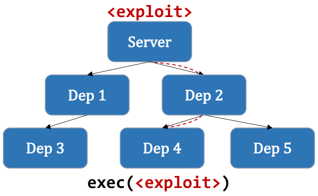


Figure 1. Arrows represent a *depends on* relationship. A victim application (Server) passes attacker-controllable input (`exploit`) to its vulnerable dependency, Dep 4 (dataflow indicated by dashed red arrows).

Real-world attacks. In a real-world attack scenario (e.g., Figure 1), a Node.js package that unsafely uses sensitive Node.js APIs is included as a *dependency* of a *victim* application. An attacker, Atk, can be any user communicating with the victim application through its public interface. Atk-controlled input is passed, unsanitized[1], from the victim application to the dependency’s public API. We consider an attack to be successful if a payload from Atk’s input is included in arguments to sensitive API (e.g., `exec` [69]) calls made by the dependency.

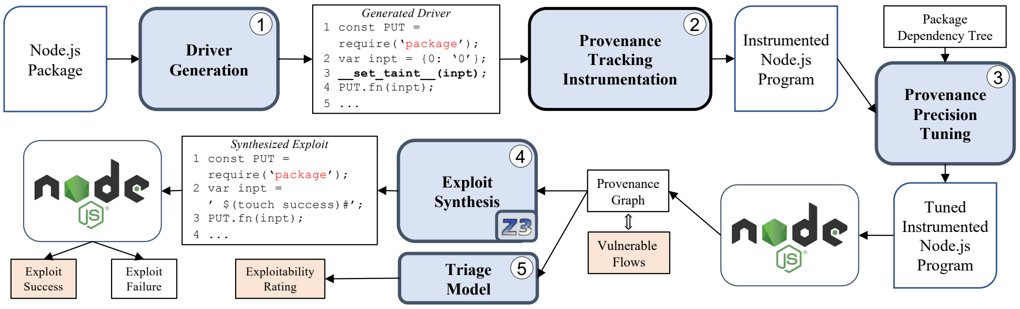
Attacker model. We utilize an idealized model of the above scenario that echoes trust assumptions in prior work [91]. In our attacker model, Atk *directly* passes input to the dependency. We consider *all* public APIs of the dependency to be the attack surface of the package. This model assumes that every public API can realistically be called with attacker-controllable input. Like prior work, we rely on this assumption because realistic usage is difficult to predict; it is safer to over-approximate possible attacks. Finally, we scope this work to focus on two types of severe attacks: arbitrary code execution (ACE) [18] and arbitrary command injection (ACI) [17]. An attacker capable of these can launch other attacks, e.g., directory traversal [19], by extension.

2.2. NodeMedic Overview

NodeMedic is an end-to-end infrastructure for discovery and analysis of ACE and ACI vulnerabilities in Node.js packages. NodeMedic consists of a multistage pipeline (Figure 2) that includes driver generation (①), provenance analysis (②), precision tuning (③), and automated confirmation (④) and triage (⑤) of vulnerabilities. We detail the provenance analysis first and then describe how it fits into the end-to-end pipeline.

NodeMedic’s provenance analysis. The infrastructure applies source-to-source rewriting of a Node.js package’s (and dependencies’) source code using Jalangi2 [82], [83], modified to support ECMAScript 6+ features (Section 3.4). Jalangi2 inlines instrumentation hooks that the analysis interfaces with (Section 3.1) to track data *provenance*, which is an extension of binary taint where a history of operations on the data is also maintained (Section 3.2). The analysis associates *provenance nodes*

[1] Dependencies typically provide sanitization. Packages requiring caller sanitization are marked as false positives (Section 5.5).



```
const PUT =
  require('package');
var inpt = {0: '0'};
__set_taint__(inpt);
PUT.fn(inpt);
...
```

```
const PUT =
  require('package');
var inpt =
  ' $(touch success)#';
PUT.fn(inpt);
...
```

Figure 2. NODEMEDIC system diagram. Blue-shaded, numbered components are pieces of the infrastructure. Orange-shaded boxes are key outputs.

with each object and primitive value (via wrapping with proxies [4], [95]).

Since NODEMEDIC focuses on analyzing public APIs (Section 2.1), the taint sources—variables holding potentially attacker-controlled data—are inputs to these APIs. Optionally, analysts can annotate additional sources like network or file-system sources (Section 3.2.1). Direct taint flows in operations such as variable assignment, primitive operations, and function calls propagate provenance according to *propagation policies* (Section 3.2.2).

Detecting vulnerable provenance flows consists of checking whether data associated with a tainted provenance node reaches a *sink*. ACE vulnerabilities' sinks are `eval` [63] and the `Function` constructor [64], both of which allow for execution of code. ACI vulnerabilities' sinks are `exec` [69] and `execSync` [71], both of which allow for a new process to be spawned. Beyond this predefined list of common sinks, analysts can specify custom sinks via annotation (Section 3.2.1). Upon reaching a sink with tainted data, a *provenance graph* is produced that can be used for further analysis (Section 4).

End-to-end infrastructure. We describe NODEMEDIC's end-to-end analysis infrastructure (Figure 2) using a case study, *font-converter*. It is a Node.js package with more than 7.8K downloads that provides a JavaScript interface to the tool FontForge for font file format conversion; it can be imported by a parent web application server to convert a user-provided font file. NODEMEDIC identifies and confirms an ACI vulnerability in *font-converter*. Through our reporting, it was assigned a CVE and a CVSS score of 9.8 [89], indicating critical severity.

The input of NODEMEDIC is a list of one or more Node.js packages to be analyzed; to analyze *font-converter*, an analyst provides NODEMEDIC the package name and version. NODEMEDIC automatically downloads and installs *font-converter* from npm in a sandboxed environment, and automatically generates a *driver* program (Section 4.1) that will import *font-converter* and execute its public APIs with values, passed for all arguments, that are marked as tainted (potentially attacker-controllable).

During provenance analysis instrumentation, dependencies can be tuned for over-approximated analysis (Section 3.3). NODEMEDIC then executes the instrumented code with off-the-shelf Node.js, which completes in 0.6 seconds, and outputs potentially vulnerable flows from tainted inputs to sinks as a provenance graph (Appendix Figure 18). In the provenance graph, leaf nodes are program inputs or constants. The remaining nodes are operations that data passes through, terminating at the sink, `exec`. The flow of tainted data is indicated by red edges. We detail the construction and interpretation of provenance graphs in Section 3.2.1.

```
var PUT = require('font-converter');
var x = "$(touch success);# ";
try { new PUT(x,x,x,x); }
catch (e) { console.log(e); }
```

Figure 3. Auto-generated *font-converter* exploit driver.

```
var command = 'fontforge -script "' +
  forgeScriptPath + '" "'+src+'" "'+dst+'"'
exec(command, callback);
```

Figure 4. Vulnerable code snippet from *font-converter*.

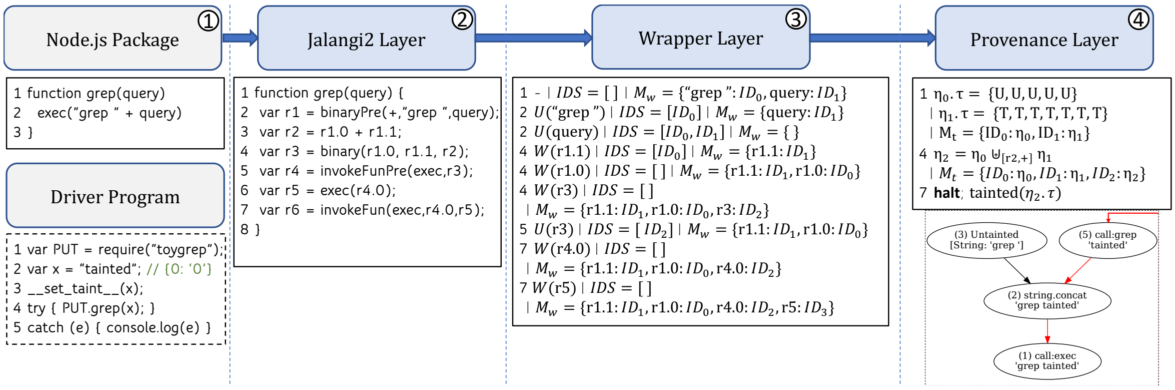
Provenance graphs can be used for further automated analysis. We define two: 1) Synthesizing a candidate exploit for a potentially vulnerable flow (Section 4.2). The candidate exploit is executed to test whether the flow is exploitable. 2) Rating the exploitability of a flow (Section 4.3). Operations in the provenance graph naturally match actions of an attacker (providing input) and defender (e.g., sanitization). We can use Attack-defense trees [45] to probabilistically model exploitability. The derived ratings can help prioritize the order in which tainted flows are reviewed or fixed by an analyst.

NODEMEDIC analyzes the provenance graph with its triage rating model and predicts that the flow is highly exploitable, signaling to analysts its high priority for review and mitigation. Next, NODEMEDIC synthesizes a candidate exploit, generates a driver to call the package with the exploit (Figure 3), and executes it. NODEMEDIC checks for the desired effect of the exploit (creation of the file `success`) and finds that it was successful. NODEMEDIC outputs a report for *font-converter*, indicating the discovered flow and successful exploit.

The analyst can review the provenance graph and find that the call to FontForge happens via execution of the Node.js `exec` API [69] with a shell command that interpolates *font-converter*'s `src` argument (path to the font file to convert) without sanitization (Figure 4). The possibility of a vulnerability is confirmed by the candidate exploit NODEMEDIC generated (Figure 3); if *font-converter* is passed a user-controllable font file name, then a user can launch an ACI attack.

3. Provenance Analysis Methodology

The provenance analysis (Figure 2, component ②) of NODEMEDIC adopts a layered architecture that separates provenance label bookkeeping instrumentation (*Jalangi2* and *wrapper* layers; Section 3.1) from provenance tracking and graph construction (*provenance layer*; Section 3.2). We explain the architecture and then discuss



```
1 function grep(query)
2   exec("grep " + query)
3 }
```

```
1 var PUT = require("toygrep");
2 var x = "tainted"; // {0: '0'}
3 __set_taint__(x);
4 try { PUT.grep(x); }
5 catch (e) { console.log(e) }
```

```
1 function grep(query) {
2  var r1 = binaryPre(+,"grep ",query);
3  var r2 = r1.0 + r1.1;
4  var r3 = binary(r1.0, r1.1, r2);
5  var r4 = invokeFunPre(exec,r3);
6  var r5 = exec(r4.0);
7  var r6 = invokeFun(exec,r4.0,r5);
8 }
```

1 - | $IDS = [\,]$ | $M_w = \{\text{“grep ”}{:}\,ID_0, \text{query}{:}\,ID_1\}$
2 $U(\text{“grep ”})$ | $IDS = [ID_0]$ | $M_w = \{\text{query}{:}\,ID_1\}$
2 $U(\text{query})$ | $IDS = [ID_0, ID_1]$ | $M_w = \{\,\}$
4 $W(\text{r1.1})$ | $IDS = [ID_0]$ | $M_w = \{\text{r1.1}{:}\,ID_1\}$
4 $W(\text{r1.0})$ | $IDS = [\,]$ | $M_w = \{\text{r1.1}{:}\,ID_1, \text{r1.0}{:}\,ID_0\}$
4 $W(\text{r3})$ | $IDS = [\,]$
 | $M_w = \{\text{r1.1}{:}\,ID_1, \text{r1.0}{:}\,ID_0, \text{r3}{:}\,ID_2\}$
5 $U(\text{r3})$ | $IDS = [\,ID_2]$ | $M_w = \{\text{r1.1}{:}\,ID_1, \text{r1.0}{:}\,ID_0\}$
7 $W(\text{r4.0})$ | $IDS = [\,]$
 | $M_w = \{\text{r1.1}{:}\,ID_1, \text{r1.0}{:}\,ID_0, \text{r4.0}{:}\,ID_2\}$
7 $W(\text{r5})$ | $IDS = [\,]$
 | $M_w = \{\text{r1.1}{:}\,ID_1, \text{r1.0}{:}\,ID_0, \text{r4.0}{:}\,ID_2, \text{r5}{:}\,ID_3\}$

1 $\eta_0.\tau = \{U, U, U, U, U\}$
 | $\eta_1.\tau = \{T, T, T, T, T, T, T\}$
 | $M_t = \{ID_0{:}\,\eta_0, ID_1{:}\,\eta_1\}$
4 $\eta_2 = \eta_0 \uplus_{[r2,+]} \eta_1$
 | $M_t = \{ID_0{:}\,\eta_0, ID_1{:}\,\eta_1, ID_2{:}\,\eta_2\}$
7 **halt**; $\text{tainted}(\eta_2.\tau)$

Figure 5. NODEMEDIC provenance analysis layers and example. Line numbers in the wrapper and provenance layers correspond to line numbers from the Jalangi2 layer. Panels are indicated by circled numbers.

key features that enable scalability: tunable propagation policy precision (Section 3.2.2) with automatic tuning of dependency analysis precision (Section 3.3); and compatibility: support for ES6+ (Section 3.4).

Architecture overview. We demonstrate the architecture via a running example (Figure 5): A toy package that exports a function called `grep` that takes a `query` and executes a shell command to run the grep command line utility on the `query` without sanitization. While simplistic, this is conceptually similar to the vulnerable code of *font-converter*. Figure 5 is split into four panels. ①: Code for the toy package and driver program (dashed border) that executes it with a tainted string, `"tainted"` passed as the `query` argument. ②: Idealized result of Jalangi2 layer instrumentation on the package code. ③: Runtime state of the wrapper layer at relevant lines of the Jalangi2 layer. ④: Runtime state of the provenance layer and a subset of the generated provenance graph (dashed border).

3.1. Instrumentation for Provenance Tracking

The first layer of the provenance analysis, the *Jalangi2 layer*, provides instrumentation hooks that the second layer, the *wrapper layer*, uses to assign unique identifiers to primitive values such as strings.

Jalangi2 layer instrumentation. NODEMEDIC utilizes Jalangi2 [82], [83] for source-to-source rewriting that injects instrumentation hooks into JavaScript code, as seen on lines 2, 4, 5, and 7 of Figure 5, panel ②. These hooks enable our analysis to execute arbitrary JavaScript code before and after each operation of the instrumented program, and thereby implement the semantics of the subsequent layers (panels ③, ④). However, not all code is instrumented by Jalangi2; uninstrumented code appears in the following two ways. 1) Native (built-in) functions which are implemented internally in C++. 2) In dependencies manually marked to not be instrumented by NODEMEDIC users. Within uninstrumented code we cannot track provenance. This is a source of imprecision that is handled via propagation policies (Section 3.2.2).

Wrapped primitive values via proxies. Tracking taint for primitive values is critical for identifying vulnerabilities because primitive strings are the usual attack vector for ACE and ACI attacks on Node.js packages. However, JavaScript primitive values cannot have new properties, e.g., taint tags, attached to them, so prior work does one of the following. 1) Modify the JavaScript engine's primitive values [12], [37], [46], [47], [61]. This requires deep engine modification and is not robust to engine updates. 2) Implement shadow variables and fields [41]. Careful replication of semantics is required to keep the original and shadow copies synchronized. 3) Box primitives and attach taint as a field [14], [76], [77]. JavaScript allows code to inspect object properties (e.g., `Object.getOwnPropertyNames` and `Object.keys`); added fields can alter program semantics.

$$\begin{array}{llcl}
\textit{Unique Ids} & ID & ::= & \{\text{ctr} : number\} \\
\textit{IDStack} & IDS & ::= & \cdot \mid ID, IDS \\
\textit{Wrapper Map} & M_w & ::= & \cdot \mid Proxy(v) \rightarrow ID \\
\textit{Wrapper Expr.} & w & ::= & v \mid wrap(v) \mid unwrap(v) \\
 & & & \mid Proxy(v)
\end{array}$$

$$\boxed{IDS, M_w \triangleright w \longrightarrow IDS', M'_w \triangleright w'}$$

(WRAP-A)

$$\frac{IDS = ID :: IDS' \qquad M'_w = M_w[Proxy(v) \rightarrow ID]}{IDS, M_w \triangleright wrap(v) \longrightarrow IDS', M'_w \triangleright Proxy(v)}$$

(WRAP-B)

$$\frac{IDS = \cdot \qquad M'_w = M_w[Proxy(v) \rightarrow \text{freshID}]}{IDS, M_w \triangleright wrap(v) \longrightarrow IDS, M'_w \triangleright Proxy(v)}$$

(UNWRAP)

$$\frac{IDS' = ID_v :: IDS \qquad M'_w = M_w/[\text{Proxy}(v) \rightarrow ID_v]}{IDS, M_w \triangleright unwrap(\text{Proxy}(v)) \longrightarrow IDS', M'_w \triangleright v}$$

Figure 6. Simplified wrapping and unwrapping semantics.

In the wrapper layer, NODEMEDIC uses *proxies* [4], [95] to create boxed (*wrapped*) values; primitives are wrapped in an object. Unlike 3), NODEMEDIC maintains a key-value map, the *wrapper map*, M_w, to associate each wrapped value with a unique *identifier*. This can be seen in Figure 5 (panel ③, line 1); the strings `"grep "` (with a space) and `query` are assigned identifiers ID_0, ID_1, respectively. The provenance layer reads M_w and associates provenance data with identifiers, rather than directly to objects and wrapped values (Section 3.2), as seen in Figure 5 (panel ④, line 1).

Wrapper layer semantics. We present key syntactic constructs of the wrapper layer and rules for the wrapping and unwrapping operations in Figure 6. We write v to denote primitive (stack-allocated) values, such as strings, numbers, booleans, and symbols. Unique identifiers are denoted *ID*, which is simply an object containing a numeric field. Proxied values are denoted *Proxy(v)*. The wrapper map, M_w, maps each proxied value to an *ID*.

The rule UNWRAP is invoked before values are passed to uninstrumented code. The value's identifier is placed on the stack *IDS*. The rules WRAP-A and WRAP-B are invoked such that during re-wrapping the value receives its previous identifier from *IDS* (or a fresh identifier). The behavior of these operations on the toy example can be seen in panel ③ of Figure 5. For each relevant line of instrumented source code from panel ② we present the corresponding wrap (*W*) and unwrap (*U*) operations performed, alongside their effect on *IDS* and M_w.

Line 1 shows wrapper layer state at the start of the function. Line 2 shows the effect of UNWRAP. For example, unwrapping `"grep "` places its identifier ID_0 on *IDS* and removes it from M_w. After the concatenation is performed, on line 4 we wrap (WRAP-A) `r1.1` (previously the `query`) first, and then wrap `r1.0` (previously `"grep "`), taking its previous identifier, ID_0 from *IDS* and re-adding it to M_w. The last operation on line 4, wrapping `r3`, uses rule WRAP-B and generates a fresh identifier ID_2 for the result of the concatenation. This pattern of unwrapping and wrapping is repeated for ID_2 on lines 5 and 7, and another fresh identifier is introduced in the last wrapper-layer operation on line 7. The full semantics include bookkeeping of function stack frames to maintain consistency of *IDS* during implicit coercion.

3.2. Policy-based Taint Provenance Tracking

We describe how the *provenance layer* uses the wrapper map, M_w, to implement provenance tracking in Section 3.2.1. Then, we introduce *precision* for provenance policies and discuss policies for built-in JavaScript data types in Section 3.2.2.

3.2.1. Provenance tracking. We use an idealized JavaScript semantics, μJS, inspired by NanoJS [92] to demonstrate provenance tracking. The core syntactic elements of μJS are described below. Here, $s \in \mathit{Strings}$, $x \in \mathit{Variables}$, $f \in \mathit{Functions}$.

$$\begin{array}{llll}
\mathit{Values} & v & ::= & s \mid f \mid \{s_1 : v, \cdots, s_n : v\} \\
\mathit{Expressions} & e & ::= & v \mid x \mid e + e \mid e\,e \mid e.e \\
\mathit{Commands} & c & ::= & e \mid \texttt{var}\ x \mid x := e \\
 & & & \mid x.e := e \mid c; c
\end{array}$$

Provenance propagation is layered on top of μJS as follows. Taint tags can be boolean, b (though we will use T, U rather than T, F to indicate Tainted and Untainted), for tracking taint of structureless primitive values (e.g., numbers), or compound mappings of object properties or string indices to taint tags. In Figure 5 (panel ④, line 1) we show tags (omitting indices) for two strings, the untainted literal, `"grep "` and the tainted input, `"tainted"`. Commands are extended to include annotations for indicating tainted data and sinks.

$$\begin{array}{llll}
\mathit{Taint\ Tags} & \tau & ::= & b \mid \{s_1 : \tau, \cdots, s_n : \tau\} \\
\mathit{Taint\ Cmds} & c_t & ::= & \texttt{taint}(v) \mid \texttt{sink}(v)
\end{array}$$

The goal of provenance tracking is, for a tainted value v', to be able to identify all of the input and constant values $v_1, \ldots, v_n$, and all of the operations with which they were combined to produce v'. Considering the toy example (Figure 5, panel ①), the value reaching the `sink`, `exec`, can be described as a concatenation of the constant `"grep "` and the input `query`. This information is tracked via a richer notion of taint; *provenance nodes*, which are used to form *provenance graphs*.

Definition 3.1 (Provenance Node)**.** A provenance node is a four-tuple $\eta = (o, v, \tau, \phi)$ where o is either a taint command c_t, built-in operation op, a function f, or an object property access $e.e$, or nothing (literal); v is a JavaScript object or primitive value; τ is the taint tag for v; ϕ is a (possibly-empty) set of *parent* provenance nodes.

In the toy example, the provenance node for the untainted string `"grep "` is: ($\cdot$, `"grep "`, $\{U, U, U, U, U\}$, $\{\}$). A *taint map*, M_t, associates *IDs* (from M_w) of primitive values and objects to provenance nodes. Compound structures' (e.g., objects) fields have their own entries in M_t, so we can precisely record which fields are tainted.

$$\mathit{Taint\ Map} \quad M_t \quad ::= \quad \cdot \mid M_t,\ \mathit{ID} \mapsto \eta$$

The general form of the taint propagation judgement is $M_w, M_t \models c \hookrightarrow M_t'$. It describes an update to the taint map based on operations in c.

Provenance propagation semantics. We categorize operations that combine two or more values as *joins*. For example, the string concatenation on line 2 of the toy example (Figure 5, panel ①) is a join operation. Provenance propagation for join operations is defined as a join of provenance nodes: $\eta_1 \uplus_{[v,o]} \eta_2$. It is parameterized by the resulting value, v, and the JavaScript operation performed, o. Below is its over-approximative form, which does not apply specific policies depending on o:

$$\frac{\eta_1 = (\_, \_, \tau_1, \_) \qquad \eta_2 = (\_, \_, \tau_2, \_) \\ \eta_3 = (o, v, \tau_1 \uplus \tau_2, \{\eta_1, \eta_2\})}{\eta_1 \uplus_{[v,o]} \eta_2 \;=\; \eta_3}\ \textsc{Join}$$

The join operation is used to compute taint map updates. Considering our toy example, propagation for the concatenation operation is shown in Figure 5 (panel ④, line 4), and results ($\tau_1 \uplus \tau_2$ merges tags) in the node:

(`string.concat`, `"grep tainted"`,
$\{U, U, U, U, U, T, T, T, T, T, T, T\}$, $\{\eta_0, \eta_1\}$)

Here, η_0 and η_1 are provenance nodes for the `"grep "` and `"tainted"` strings. The taint map update for this new node is also seen on line 4 of panel ④. In practice, the implementation is optimized by storing references to existing nodes rather than duplicating them in the join.

Provenance graphs. During execution of the package, provenance is propagated until the execution terminates naturally (i.e., as it would behave without provenance tracking), or until a `sink` is reached.

$$\textsc{(Sink-Continue)}\\ \frac{M_t[M_w[v]] = (\_, \_, \tau, \_) \qquad \neg\texttt{tainted}(\tau)}{M_w, M_t \models (\texttt{sink}(v)) \hookrightarrow M_t}$$

If $\texttt{sink}(v)$ is reached with a non-tainted value, execution continues (SINK-CONTINUE). On the other hand, if $\texttt{tainted}(\tau)$ holds execution halts (e.g., Figure 5, panel ④, line 7). At that point we take the tainted node

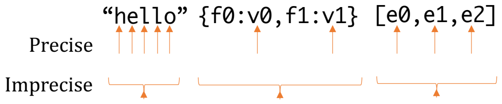


Figure 7. Precise and imprecise policies for strings, objects, and arrays. Orange arrows indicate what is assigned a taint tag for each policy type.

$\eta = (\_, \_, \_, \phi)$ and transform it into a provenance graph by recursively associating an edge ω_i between η and each parent node $\eta_{p_i} \in \phi$. In other words, the provenance graph will link together all of the provenance nodes generated along the path to the sink into a single graph.

Definition 3.2 (Provenance Graph)**.** A provenance graph is a directed acyclic graph $G = (\mathcal{N}, \mathcal{E})$ where $\eta \in \mathcal{N}$ is a provenance node; $\omega \in \mathcal{E}$ is a directed edge indicating taint propagation from η_1 to η_2, where $\eta_1, \eta_2 \in \mathcal{N}$.

Provenance graph example. A provenance graph is generated by analyzing package execution with a tainted string, `"tainted"`, (Figure 5, panel ④, lower box). The graph illustrates how input passes through the `grep` function call (node 5) and is concatenated with the string `"grep "` (node 2) before reaching the sink, `exec` (node 1). A vulnerability is apparent: an exploit payload can be passed via the `query` input.

3.2.2. Propagation policies.

The provenance analysis relies on *propagation policies* to specify propagation for native (uninstrumented) data types' operations. We demonstrate precision levels of common data types in Figure 7: For compound structures such as strings, objects, and arrays the analysis provides precise propagation: character-level for strings, field-level for objects, and element-level for arrays; or imprecise propagation: at the whole string, object, or array-level. More precise policies lead to less over-tainting and fewer false positive reports, but higher analysis runtime. NodeMedic's policies are summarized below, with details for all policies in Appendix A.

Objects. NodeMedic uses two invariants to relate precise object-level and field-level information: 1) If `obj` is tainted, then its fields are tainted; intuitively, if `obj` is attacker-controlled, then its fields are too. 2) If all `obj` fields are tainted, then `obj` is tainted; if an attacker controls every field, then they control the entire object.

Strings. NodeMedic provides precise taint models for common string operations and a method of encoding taint information directly into a string using unused bits in the unicode representations of characters, allowing taint to automatically propagate for some operations (Appendix B).

Function calls. *Imprecise*: The return value of a function call, if any, receives the join of call arguments' provenance nodes (see $\geq$-CALL in Section 3.3). *Precise*: Provenance is propagated via the function body's statements.

User-defined policies for compositional analysis. NodeMedic allows module-specific propagation policies, *module policies*, to be added. This is desirable for compositional analysis where these policies serve as summaries for APIs in large modules. For example, we define policies for the popular library *lodash*'s commonly used `for` and `forEach` functions, allowing precise propagation without full instrumentation of *lodash* (26K LoC).

Algorithm 1 Dependency Propagation Policy Tuning

```
 1: $\mathcal{F} \leftarrow \{\cdot\}$, $\psi \leftarrow$ Root($t_d$)
 2: if Size($t_d$) $\geq \mathtt{m_{deps}}$ then
 3:     Tune($\psi$, 0)
 4: end if
 5: procedure Tune($\psi$, $d$)
 6:     for $f \in$ Exported($\psi$) do
 7:         $\mathcal{F} \leftarrow \mathcal{F}, (f \mapsto \neg(d \geq \mathtt{m_{depth}}))$
 8:     end for
 9:     for $\psi' \in$ Children($\psi$) do
10:         Tune($\psi'$, $d + 1$)
11:     end for
12: end procedure
```

3.3. Auto-tuning of Propagation Policies

While NodeMedic is an offline analysis, performance does aid scalability. Provenance tracking incurs additional analysis time compared to traditional taint tracking (Section 5.3). To remain performant for Node.js packages with tens to hundreds of dependencies (Appendix Figure 17), we apply the observation that not every dependency needs precise analysis [67]. We define an algorithm for *auto-tuning* of propagation policies (Figure 2, component ③) that decides what subset of the dependencies will remain uninstrumented and be replaced by an over-approximative policy. We first describe how we compute a package's dependency tree, and then introduce our tuning algorithm.

Dependency trees. We start from a *root* package and read the `"dependencies"` key of the package's manifest, `package.json`. We add edges from the root to each dependency. We then recurse on each dependency as the new root. This results in a *dependency graph*, which we convert to a *dependency tree*, t_d, by breaking cycles and duplicating nodes with multiple incoming edges so each node has a single incoming edge. The *size* is the number of dependencies; the *depth* is the longest path length.

Deciding propagation policies. We define an algorithm (Algorithm 1) that, given a dependency tree, t_d, produces a *marked function context* $\mathcal{F}$ that maps functions f to boolean values, i.e., $\mathcal{F} ::= \cdot \mid \mathcal{F},\ f \mapsto b$. The boolean b indicates whether precise ($b = \mathtt{true}$) or imprecise ($b = \mathtt{false}$) policies should be used in the analysis of f.

For packages with enough dependencies, the algorithm walks the dependency tree and marks dependencies below a certain depth to be over-approximately analyzed. The algorithm is parameterized by two analyst-chosen parameters: (1) the *minimum number of dependencies*, $\mathtt{m_{deps}}$ and (2) the *minimum depth*, $\mathtt{m_{depth}}$. On line 2, if the tree's number of dependencies meets $\mathtt{m_{deps}}$, the algorithm will walk the tree and track the current depth. On line 7, if the current depth at package ψ meets $\mathtt{m_{depth}}$, all of ψ's public functions f will be marked $f \mapsto \mathtt{false}$. Dependencies marked with `false` will be over-approximately analyzed as described below.

An example execution of the algorithm is shown in Appendix Figure 16. There is an alternate configuration of the second parameter: minimum depth set to *dynamic* (*dyn*). In this case, $f \mapsto \mathtt{false}$ will be applied (at varying depth) to all leaf nodes. We provide parameter selection recommendations in Section 5.4.2.

Over-approximated function analysis. We extend our provenance analysis to access the marked function context $\mathcal{F}$ during propagation. During analysis, if $f \mapsto \texttt{false}$, we *over-approximate* propagation for f by propagating purely on the basis of the provenance nodes of the arguments to f [67]. Below, we present the rule for an over-approximated call.

$$\textsc{(}\geq\textsc{-Call)}\quad \frac{\begin{array}{c} M_t[M_w[v_1]] = \eta_1 \;\dots\; M_t[M_w[v_n]] = \eta_n \\ \mathcal{F}[f] = \texttt{false} \quad \eta' = \eta_1 \uplus_{[z,f]}, \dots, \uplus_{[z,f]} \eta_n \end{array}}{\mathcal{F}, M_w, M_t \models (z := f\ v_1, \dots, v_n) \hookrightarrow M_t[M_w[z] \mapsto \eta']}$$

The provenance node assigned to the result of the function call is the join of the provenance nodes of the function call arguments. This is a safe over-approximation assuming taint commands c_t are not present in f, as is the case for taint sources in our attacker model (Section 2.1).

3.4. Supporting ECMAScript 6+

NodeMedic uses Jalangi2 [82], [83], which only supports ECMAScript 5.1. We use the transpiler Babel [5] to rewrite ES6+ code for Jalangi2. We intercede Babel in Jalangi2's instrumentation process; before Jalangi2 attempts to parse the program, we transpile that program into ECMAScript 5.1. However, this is not sufficient to support new APIs such as promises, maps, and sets.

Promises and promisify. Provenance propagation for promises can be separated into three cases: 1) Within promise bodies; 2) Upon promise resolution or rejection; 3) During promise function wrapping (*promisify*). NodeMedic handles case 1 as regular function bodies. Case 2 cannot be handled by NodeMedic or other instrumentation-based work because `resolve` and `reject` are native, anonymous functions. To support case 3, NodeMedic implements a generalized policy: *sink propagation*. Sink propagation works as follows: If f is an uninstrumented function that accepts a sink as an argument and returns a function f', then f' is a sink. Support for this allows NodeMedic to find flows that otherwise would be missed (Section 5.5).

Maps and sets. NodeMedic supports both precise and imprecise policies for maps and sets that mirror array policies. *Imprecise*: A single provenance node is used for the entire map (or set). *Precise*: Individual elements of the map (or set) have their own provenance nodes.

4. End-to-End Analysis Methodology

In order to execute packages with provenance analysis we implement automated package setup and driver generation (Figure 2, component ①), explained in Section 4.1. Using the analysis output, provenance graphs, we implement two post-analyses: exploit synthesis (component ④), described in Section 4.2, and triage rating (component ⑤), described in Section 4.3.

4.1. Automated Setup and Driver Generation

NodeMedic applies a lightweight pre-analysis to generate a JavaScript program (driver) that can execute the package. The core technique is to import the package and enumerate all properties defined on its exported interface. We gather properties that correspond to functions and constructors and extract formal parameters, including rest parameters. Since JavaScript is dynamically typed we do not determine their types.

A driver is generated (Figure 5, panel ①, dashed box) that executes each exported function and constructor with the appropriate number of arguments (line 4), automatically annotated as taint sources (line 3). For maximum compatibility, the driver provides as arguments (line 2) an object with a single property, `0`, which maps to a string[2]. This leverages JavaScript's generous coercion features; it is coerced to a string if used in string operations and can be indexed like an array due to the property `0`.

4.2. Exploit Synthesis with Provenance Graphs

Provenance analysis can produce numerous potentially vulnerable flows which must be manually reviewed to confirm true positives. We reduce this burden by deriving SMT constraints from provenance graphs to synthesize candidate ACE and ACI exploits and automatically check if flows are exploitable. We explain each step using the toy example's provenance graph (Figure 5, panel ④).

Processing provenance graphs. ACE and ACI are typically string-based exploits. From a provenance graph, we extract nodes concerning strings (s): 1) untainted constant string literals; 2) tainted input strings, treated symbolically; 3) operations that join multiple s ($\boxplus$); 4) operations that modify strings: functions with manually modeled taint propagation f_m, and automatically precisely, f_p, and imprecisely, f_i, taint propagated functions; 5) sinks, f_s.

Below, we define an *operation tree* to store this information. We use a context, Γ, to store mappings between strings and taint types, ρ; T: Tainted; U: Untainted.

$$\begin{array}{llcl} \textit{Taint types} & \rho & ::= & T \mid U \\ \textit{String context} & \Gamma & ::= & \cdot \mid \Gamma,\ s \mapsto \rho \\ \textit{Functions} & f & ::= & f_m \mid f_p \mid f_i \mid f_s \\ \textit{Operation tree} & t_o & ::= & s \mid t_o \boxplus t_o \mid f\ t_{o_1} \cdots t_{o_n} \end{array}$$

Rules for transforming a provenance node to an operation tree are of the form $\Gamma \triangleright \eta \Rightarrow \Gamma', t_o$. Below we present an example transformation for a string concatenation node.

$$\frac{\begin{array}{l} \phi = \{\eta_1, \eta_2\} \quad \Gamma \triangleright \eta_1 \Rightarrow \Gamma_1, t_{o_1} \\ \Gamma \triangleright \eta_2 \Rightarrow \Gamma_2, t_{o_2} \quad \Gamma' = \Gamma_1 \cup \Gamma_2 \end{array}}{\Gamma \triangleright (+, \_, \_, \phi) \Rightarrow \Gamma', t_{o_1} \boxplus t_{o_2}}\ \textsc{(Transform-+)}$$

This rule transforms instances of concatenation operations into operation trees by recursively transforming each parent node into an operation subtree and emitting a new $\boxplus$ node that joins the subtrees.

Toy Ex. Step 1: The operation tree is $t_o =$ (`exec` ("grep " $\boxplus$ "tainted")) with string context $\Gamma = \{$"grep " $\mapsto U$, "tainted" $\mapsto T\}$.

Deriving SMT constraints. Synthesizing a candidate exploit requires finding attacker-controllable inputs that result in an exploit string, v_{ATK}, reaching the `sink`. Given

[2] In the toy example, we use the string "tainted" for illustrative purposes; the actual argument can be seen in the adjacent comment.

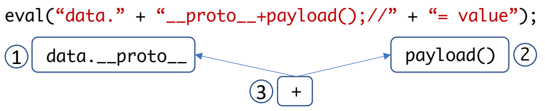


Figure 8. Example evaluation of an ACE concrete exploit string.

t_o, we generate an SMT formula to solve for assignments to SMT string constants that will make up v_{ATK}. This is done via a rewriting system that transforms elements of t_o to SMT statements.

The subset of SMT grammar we are concerned with consists of SMT statements z which are either string literals s_z, SMT operations o_z, or string constants s_c. Given $s \in t_o$, the rewriting rules query Γ to translate s either to a literal ($\Gamma[s] = U$) or constant ($\Gamma[s] = T$). We maintain a context β that stores declarations of SMT constants. The form of the rewriting judgement is $\Gamma, \beta \triangleright t_o \Rightarrow \beta', z$. We present the rule for concatenation below.

$$\frac{\begin{array}{c}\Gamma, \beta \triangleright t_{o_1} \Rightarrow \beta_1, z_1 \quad \Gamma, \beta \triangleright t_{o_2} \Rightarrow \beta_2, z_2 \\ \beta' = \beta_1 \cup \beta_2\end{array}}{\Gamma, \beta \triangleright (t_{o_1} \boxplus t_{o_2}) \Rightarrow \beta', (\texttt{str.++}\ z_1\ z_2)}\ (\textsc{Rewrite-}\boxplus)$$

As we see above, an instance of $\boxplus$ is rewritten by recursively rewriting each subtree t_{o_i} into SMT formulae that are given as arguments to the Z3 `str.++` method. Sink functions map to the Z3 `str.contains` method, called with the exploit string, v_{ATK}. Mappings of other functions are done per-function. For example, the `Array join` function ($\in f_m$) is modeled via successive concatenation of the (literal or constant) elements of the array.

Toy Ex. Step 2: t_o is rewritten (using Γ), resulting in $z = (\texttt{str.contains}\ (\texttt{str.++}$ "grep " $\texttt{i0})\ v_{\text{ATK}})$ with $\beta' = \{\texttt{i0}\}$.

Concrete exploit strings. The formula z contains constraints from operations but v_{ATK} needs to be instantiated with a concrete exploit, which we construct next. The attacker's input typically ends up interpolated into a concrete string, $s_0 + v_{\text{ATK}} + s_1$, during package execution. For example, for ACI with `exec`, s_0 may be the command to run and s_1 stores command flags. Executing an exploit payload amounts to constructing v_{ATK} such that v_{ATK} *completes* the prefix s_0 and *obviates* the suffix s_1 [50].

We treat v_{ATK} as a compound string consisting of $s_{\texttt{pre}} + s_{\texttt{pay}} + s_{\texttt{suf}}$. The goal of $s_{\texttt{pre}}$ is to complete s_0. The goal of $s_{\texttt{pay}}$ is to deliver the exploit payload. Finally, the goal of $s_{\texttt{suf}}$ is to cause the suffix s_1 to not be executed. Selections of $s_{\texttt{pre}}$, $s_{\texttt{pay}}$, and $s_{\texttt{suf}}$ are dictated by the vulnerability type and sourced from known exploits. Below, we use $f_\triangledown$ to indicate a fresh global function with a stateful effect (e.g., print to `stdout`), and $s_\triangledown$ to indicate a shell command with a stateful effect (e.g., file creation). Stateful effects indicate attack success and ATK's ability to execute arbitrary code or commands.

For ACI: $s_{\texttt{pre}}$ = " ", an empty space that separates the payload from prior commands; $s_{\texttt{pay}}$ = "$s_\triangledown$", in practice, we use the command "touch" with a file path "success", wrapped in evaluation; "$(touch success)", which results in file creation and indicates ATK can execute commands; $s_{\texttt{suf}}$ = "#", a bash comment delimiter to prevent execution of s_1 after the payload. For ACE: $s_{\texttt{pre}}$ = "__proto__", prototype access, which is defined for almost all objects; $s_{\texttt{pay}}$ = " + ($f_\triangledown$ ·);", addition with RHS calling $f_\triangledown$, which prints a string to `stdout` and indicates ATK can execute code. $s_{\texttt{suf}}$ = "//", a JavaScript comment delimiter to prevent execution of s_1.

An evaluation of an ACE exploit string is in Figure 8; first evaluated is the LHS (box 1) which is a valid access, next the RHS, `payload()`, which represents an instance of $f_\triangledown$ (box 2), and finally the addition operation (box 3). Nothing else is evaluated due to the comment delimiter.

Toy Ex. Step 3: z is augmented with a concrete ACI exploit string; $z = (\texttt{str.contains}\ (\texttt{str.++}$ "grep " $\texttt{i0})$ " $(touch success);#").

Solving with Z3. We now have a final context β' and a concretized SMT statement z. Next, the string constants $s_{c_i} \in \beta'$ are each prepended to $z' = (\texttt{assert}\ z)$ with the Z3 statement: $(\texttt{declare-const}\ s_{c_i}\ \texttt{String})$.

This results in the final SMT formula z'', which is given to Z3 for solving. We use $(\texttt{get-model})$ to request assignments for s_{c_i} (if z'' is satisfiable). Each s_{c_i} assignment represents a string literal value the attacker will pass to the package for execution.

Toy Ex. Step 4: We wrap z in $z' = (\texttt{assert}\ z)$ and declare constants according to β', resulting in the SMT statement z'' (Figure 9). Finally, z'' is given to Z3, which returns the trivial satisfying assignment: `i0` = " $(touch success);#".

```
(declare-const i0 String)
(assert (str.contains
  (str.++ "grep " i0)
  " $(touch success);#"))
(check-sat)
(get-model)
```

Figure 9. Toy Example SMT statement z''

```
1  var PUT = require("toygrep");
2  var x = " $(touch success);#";
3  try { PUT.grep(x); }
4  catch (e) { console.log(e); }
```

Figure 10. Auto-generated exploit driver for the toy example.

Exploit driver. To determine exploit success we extend the package driver (Section 4.1) to invoke package APIs with the SMT-derived input and check for the desired stateful effects, e.g., file creation, to determine if the exploit payload is successfully executed. We generate different *exploit drivers* for arbitrary code execution and arbitrary command injection.

The ACE driver injects function $f_\triangledown$, defined to print a unique string to standard output. The driver attempts execution of $f_\triangledown$, detectable by monitoring process output. The ACI driver attempts execution of the shell command $s_\triangledown$ (Figure 10, lines 2-3), defined to cause file creation at a particular path, detectable by monitoring that path.

Toy Ex. Step 5: The generated exploit driver is shown in Figure 10; $s_\triangledown$ is in the generated payload `i0`.

Running the exploit driver. After the driver executes each package API with the exploit string, a check is performed to determine if exploit's effect was detected. If so, the exploit is confirmed. If all APIs execute without a detected effect, the exploit could not be confirmed. Persistent stateful effects are cleaned up between runs.

Toy Ex. Step 6: The exploit driver is executed and creation of the file, success, is observed, confirming the ACI vulnerability.

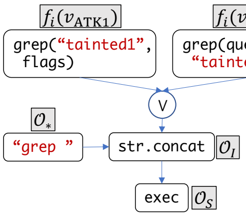


Figure 11. Extended toy example Provenance-AD-tree.

4.3. Triage Ratings with Provenance Graphs

Next, we use provenance graphs to characterize the *exploitability* of a tainted flow. Exploitability means "ease of exploitation"; a high-exploitability flow is easier for an attacker to exploit, e.g., unsanitized dataflow to a sink. Exploitability ratings can provide analysts a way to prioritize review of discovered flows, *before* exploits are manually confirmed. Prioritization is often necessary for organizations with limited resources for timely security review; attacks may happen via "weakest links" (i.e., easiest to exploit dependencies) first [25]. Exploitability ratings can be viewed as pre-confirmation analogues of CVSS scores [21], but they are not the same. CVSS scores measure severity of a confirmed exploit; a difficult exploit (low-exploitability) could be high-severity.

Provenance Attack-defense trees. We compute exploitability by transforming provenance graphs into Attack-defense trees (AD-trees) [2], [29], [34], [45], [100], modeling probabilistic attacker and defender actions. An AD-tree t is specified inductively as $t ::= p \,|\, t \wedge t \,|\, t \vee t \,|\, {\sim} t$ where A_a are attacker actions, A_d are defender actions, and $p \in A_a \cup A_d$.

In our setting, the attacker's action set, A_a, is a single action: providing input v_{ATK} to a package public API, f_i. Defender's actions, A_d, consist of JavaScript operations: $\mathcal{O}_I$: Built-in operations, e.g., `substr`. $\mathcal{O}_E$: Sanitization operations, e.g., `escape`. $\mathcal{O}_A$: Object field accesses. $\mathcal{O}_S$: Sinks. $\mathcal{O}_\star$: All other operations (unmodeled) and literals.

A provenance graph is transformed into a Provenance-AD-tree by mapping provenance nodes $\eta = (o, v, \tau, \phi)$ to Provenance-AD-tree nodes $p_\eta = (p, v, \tau)$ where $p \in A_a \cup A_d$. Tainted nodes from the parent nodes $\phi = \{\eta_1, \ldots, \eta_n\}$ represent alternative ways an attacker's input can reach a program point, thus naturally represented as a disjunction of nodes, $p_{\eta_1} \vee \cdots \vee p_{\eta_n}$. Operations p link to previous p_η, as in the provenance graph.

Definition 4.1 (Provenance-AD-tree)**.** A Provenance-AD-tree t_p is specified inductively as $t_p ::= p_\eta \,|\, t_p \vee t_p$ where $A_a = \{f_i(v_{\text{ATK}})\}$, $A_d = \mathcal{O}_I \cup \mathcal{O}_E \cup \mathcal{O}_A \cup \mathcal{O}_\star$, and $p_\eta = (p, v, \tau)$ with $p \in A_a \cup A_d$.

To illustrate, in Figure 11 we show a Provenance-AD-tree for an extended version of the toy example (Figure 5, panel ①) where `grep` now takes a second argument, `flags`, that will be passed to the grep utility. Attacker and defender actions are shown with their code and labeled with their category (in A_a and A_d, respectively).

Probabilistic ATK model. Modeling attacker behavior with AD-trees has been done with temporal automata [29], [34] and Markov chains [2]; given the fixed probability of success of each action, a graph is produced linking defender actions and attacker actions. The probabilities are typically expert-provided, however, this can be error-prone and difficult to validate [2], [29]. We also construct a (probabilistic) automata, but we instead derive probabilities from the Provenance-AD-tree structure.

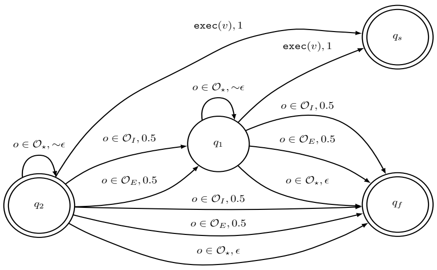

$\texttt{exec}(v), 1$
$\texttt{exec}(v), 1$
q_s
$o \in \mathcal{O}_\star, {\sim}\epsilon$
$o \in \mathcal{O}_I, 0.5$
$o \in \mathcal{O}_\star, {\sim}\epsilon$
$o \in \mathcal{O}_I, 0.5$
q_1
$o \in \mathcal{O}_E, 0.5$
$o \in \mathcal{O}_E, 0.5$
$o \in \mathcal{O}_\star, \epsilon$
q_2
$o \in \mathcal{O}_I, 0.5$
q_f
$o \in \mathcal{O}_E, 0.5$
$o \in \mathcal{O}_\star, \epsilon$

Figure 12. Example ATK probabilistic automata.

We use the automata to simulate the effectiveness of attacker's submitted inputs on the package (i.e., A_a), given the defender actions in the Provenance-AD-tree, (i.e., A_d). Intuitively, the automata is constructed such that starting states correspond to original attacker inputs, and defender actions result in transitions with some probability to states representing modifications of the attacker's input, with final states representing attacker input reaching a sink and successfully deploying a payload or failing.

We model the attacker's input via *families of strings*, $\mathcal{V}_1, \ldots, \mathcal{V}_n \in \bar{\mathcal{V}}$. Each family represents different exploit payloads that rely on different operations. For the extended toy example, $|\bar{\mathcal{V}}| \leq 2$ due to the two-subtree disjunction node (Figure 11). An automata suitable for that Provenance-AD-tree is in Figure 12. The automata starts at q_2, representing two $\mathcal{V}$. It probabilistically remains there while receiving $o \in \mathcal{O}_*$ until a built-in (O_I) or sanitizing (O_E) operation is received. At that point it can transition to q_1 (one $\mathcal{V}$) with $pr = \frac{1}{2}$ or a final failure state (q_f). At q_2 or q_1, if the sink `exec` is received, the automata transitions to the final success state (q_s). To increase legibility, we only include transitions for mentioned operations. Details of automata construction are in Appendix C.

Probability of ATK success. Following prior work, we estimate attacker success ($Pr[E_{S_{\text{ATK}}}]$) via Bernoulli trials [34]. Intuitively, this simulates the behavior of an attacker repeatedly submitting exploit payloads to a package, and estimates the frequency of successful exploits, i.e., those remaining intact through the package operations. We traverse the Provenance-AD-tree and feed defender actions as input to the automata until it halts at a final success or failure state, and count successes.

We instantiate the automata given a particular t_p, ϵ. We then randomly select an attacker-controllable node from t_p and begin a graph traversal of t_p, following the direction of taint propagation. At each step, p from the current $p_\eta = (p, v, \tau)$ is given as input to the automata. This process continues, randomly selecting a next p'_η from the set of p'_η in the valid (p_η, p'_η) edges of t_p, until the automata either halts at q_s or q_f. The frequency of halting at q_s estimates the probability of ATK success (Appendix Algorithm 2).

Triage ratings. Once $Pr[E_{S_{\text{ATK}}}]$ has been estimated, we bucket the result into three intervals: "Low": $[0, \frac{1}{3}]$; "Medium": $(\frac{1}{3}, \frac{2}{3}]$; "High": $(\frac{2}{3}, 1]$. These ratings can be used to inform a priority queue (High → Low) ordering the sequence in which potentially vulnerable flows are reviewed by an analyst.

5. Evaluation

We first describe our setup and the datasets used (Section 5.1), and then explore the following questions:
RQ1: Precision compared to prior work (Section 5.2).
RQ2: Performance compared to prior work (Section 5.3).
RQ3: Precision of propagation policies (Section 5.4.1).
RQ4: Efficacy of auto-tuning precision (Section 5.4.2).
RQ5: Discovery of new vulnerabilities (Section 5.5).
RQ6: Efficacy of exploit synthesis (Section 5.5.1).
RQ7: Triage rating safety and precision (Section 5.5.2).

5.1. Evaluation Setup and Datasets

Setup. All evaluations were performed within Docker containers running on a Ubuntu machine with an Intel Core i9-9900K@3.60GHz and 128GB of memory (though memory utilization is around 2GB per analysis container). The total analysis timeout is 15 minutes. For exploit synthesis, the Z3 timeout is 60 seconds.

Prior work dataset. We gather a set of packages with analysis results presented by prior work. We attempt to replicate the evaluation of every package analyzed by both Ichnaea [41] and Synode [91]. However, this was not possible for all packages; some no longer exist or are incompatible with newer versions of Node.js, while others use input sources, e.g., command-line, outside the threat model (Section 2.1). We are left with 21 packages.

10K package dataset. We gather a set of 10K packages from the npm repository in January, 2022 with the goal of selecting a wide, unbiased sample. Querying the npm index API provides an alphabetically-ordered list of registered packages. To avoid selection bias from similarly-named packages, we divide the index into 10 equally-sized ranges. From each range we select 1000 packages according to the following criteria. The package must: 1) Be downloadable and importable (e.g., no broken or missing packages). 2) Contain a sink checked by NODEMEDIC; lack thereof produces no true or false positives. 3) Not be client-side (e.g., no browser APIs); such packages are outside our attacker model (Section 2.1).

5.2. Precision Compared to Prior Work

We use NODEMEDIC in its default (precise) policy configuration to analyze packages analyzed by prior work. The results are presented in Table 1. In the table, TP = True positive, TN = True negative, execS = `execSync`, execFS = `execFileSync`, SY = Synode [91], and IC = Ichnaea [41]. NODEMEDIC does not introduce false negatives, matching prior work's results in every case, and for the package *systeminformation*, finding a true vulnerability that past work missed. Ichnaea described that package as a true negative, however, a vulnerability affecting that version of the package has since been discovered [88]. NODEMEDIC correctly reports a taint flow for this vulnerability. Triage and exploit synthesis results for these packages are shown in Appendix D.1.

Result 1: NODEMEDIC's provenance analysis is sufficiently precise to find vulnerabilities uncovered by prior work, as well as a vulnerability missed by prior work.

TABLE 1. ANALYSIS OF PACKAGES FROM PRIOR WORK

Package	Version	LoC	Sink	Result	Source
fish	0.0.0	55	exec	TP	SY
git2json	0.0.1	228	exec	TP	SY
gm	1.20.0	3517	exec	TP	SY
growl	1.9.2	323	exec	TP	SY
kerb_request	0.0.2	30716	exec	TP	SY
m-log	0.0.1	1164	eval	TP	SY
mixin-pro	0.6.6	449	eval	TP	SY
mobile-icon-resizer	0.4.2	4648	eval	TP	SY
mol-proto	0.0.15	5983	eval	TP	SY
mongo-parse	1.0.5	1835	eval	TP	SY
mongoosemask	0.0.6	34259	eval	TP	SY
mongoosify	0.0.3	26365	eval	TP	SY
node-libnotify	1.0.3	78	exec	TP	SY
node-os-utils	1.0.7	1097	exec	TP	IC
node-wos	0.2.3	535	execS	TN	IC
office-converter	1.0.2	113	exec	TP	IC
osenv	0.1.5	266	exec	TN	IC
pidusage	1.1.4	526	exec	TP	IC
pomelo-monitor	0.3.7	259	exec	TP	IC
system-locale	0.1.0	61	execFS	TN	IC
systeminformation	3.42.4	22102	exec	TP*	IC

TABLE 2. NODEMEDIC PERFORMANCE

Configuration	Runtime (s)			vs. Baseline	vs. Jalangi2
	min	avg	max	avg	avg
Baseline	0.03	0.04	0.09s	-	-
Jalangi2	0.27	0.46	1.15s	12.1x	-
NODEMEDIC	0.27	0.79	3.47s	20.3x	1.68x

5.3. Performance Compared to Prior Work

NODEMEDIC is intended to be run offline; it does not have strict performance requirements. Nonetheless, performance is tightly connected to scalability. We record execution time (mean of 10 runs after 3 warmup runs) of analysis under three configurations: 1) Baseline: Without instrumentation. 2) Jalangi2: With Jalangi2's instrumentation hooks, but without provenance analysis. 3) NODEMEDIC: With NODEMEDIC's provenance analysis (default configuration) using Jalangi2 instrumentation hooks. We list the runtimes (rounded) and relative average overhead of each configuration in Table 2.

NODEMEDIC incurs an average 1.7x runtime overhead over the inherent overhead of Jalangi2, for a total of 20x runtime overhead. This is larger than the most performant comparable dynamic analyses, e.g., the 10x overhead of Ichnaea [41]. While static and dynamic analyses face different scalability challenges, NODEMEDIC's runtime is fast compared to static Node.js analyses such as Nodest, which has a 30 minute timeout per analysis iteration [67].

Jalangi2's instrumentation hooks account for 59% of the overhead. However, NODEMEDIC's provenance tracking does not inherently depend on Jalangi2; it uses the hooks to inject provenance tracking logic.

Result 2: Precise provenance tracking incurs an average 1.7x slowdown over Jalangi2 for a total 20x slowdown.

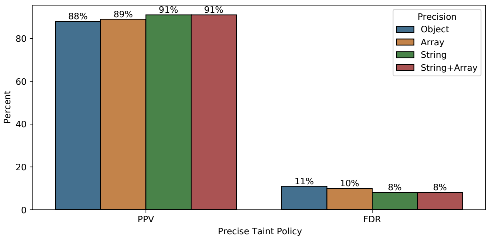


Figure 13. Precision (PPV) and false discovery rate (FDR) with precise string and array taint policies. PPV + FDR = 99% due to rounding.

TABLE 3. Policy effects on true and false positives

	+TP	-TP	+FP	-FP	Overlap w/ Object
String	1	3	0	1	23
Array	3	1	0	0	26

5.4. Tuning Propagation Policies

We use a randomly-selected quarter of our 10K dataset (termed the “baseline” set) for the following experiments.

5.4.1. Effect of precise propagation policies. We vary propagation policy precision in four experiments. (1) Just precise object tainting, which is as precise as prior work [41]; (2) Just precise string tainting; (3) Just precise array tainting; (4) The combination of (2) and (3). For each, we record true and false positives; independently validated by two security researchers. Every experiment was run without auto-tuning of policy precision. The results are presented in Figure 13.

We evaluate the precision ($PPV = \frac{TP}{TP+FP}$) and the false discovery rate ($FDR = \frac{FP}{TP+FP}$); neither measure relies on ground truth for false negatives. We find that having just object precision performs the worst; with the lowest PPV (88%) and highest FDR (11%). Using both the string and array precise taint policies at the same time performs the best; resulting in the highest PPV (91%) and the lowest FDR (8%; tied with using just the precise string policy).

In Table 3 we examine the sets of true and false positives of the string and array precise policies compared to just object precision. The precise string taint policy reduces false positives at the expense of a few true positives. The precise array taint policy is able to generally increase true positives when array operations are present.

Intuitively, imprecise policies over-taint and precise policies under-taint; however, this is not entirely true for NodeMedic, which is why we see new true positives with the precise policies. More concretely, using just object precision can result in under-tainting during string and array operations due to their native (uninstrumented) implementations; `array.push` of a tainted value would not affect taint of the array itself.

Result 3: Precise taint policies reduce false positives and increase true positives. Different policies uncover different vulnerabilities due to policy-specific models.

5.4.2. Configuring precision tuning. Automated provenance propagation precision tuning is parameterized by two variables, the *minimum depth* and the *minimum number of dependencies* (Section 3.3). In a series of experiments we independently vary these parameters and record the number of flows to sinks, the number of timeouts, and the number of vulnerabilities found, relative to the baseline (no tuning).

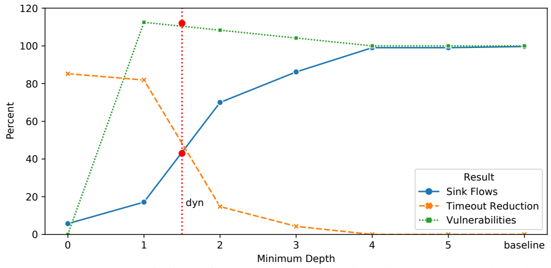


Figure 14. Percent of sink flows, reduction in timeouts, and vulnerabilities found at varying depths (x-axis) and the *baseline* (no auto-tuning). The vertical red line marks *dyn* results.

TABLE 4. Large-scale evaluation tainted flow metrics

Metric	All Avg.	TP Avg.
Last Year DLs	19585	21310
Code Size (KB)	465	502
LoC	5202	5268
Num. Deps.	5.88	6.31
Dep. Depth	1.07	1.12
Prov. Nodes	507.34	266.99
Prov. Depth	36.91	36.17
Vuln. Deps.	-	1.46
Vuln. Depth	-	0.04

As the *minimum depth* increases, more sink flows are detected at the cost of more timeouts (Figure 14). We find a balanced point—our dynamic (*dyn*)—configuration (indicated with a red dotted vertical line). Here we obtain a 41% reduction in timeouts and find the maximum number of vulnerabilities. As we increase the *minimum number of dependencies* we find there is a continuous increase in the number of timeouts and that the depth configurations' timeout rates begin to converge. Thus, an analyst using a low minimum depth (e.g., 1) should pair it with a low minimum number of dependencies.

Result 4: Automatic policy tuning's dynamic configuration balances sinks executed and timeouts to uncover the most vulnerabilities. The difference in effectiveness of each depth configuration decreases as the configured minimum number of dependencies increases.

5.5. Large-Scale Evaluation

We use NodeMedic to analyze the packages in our full 10K package dataset (Table 5). Informed by our small-scale experiments (Section 5.4), for this evaluation we configure NodeMedic to use the most precise taint policies, and use the *dyn* precision tuning setting with no minimum number of dependencies.

Out of the 10,000 packages in the dataset, 125 packages had inherent issues that prevented them from being analyzed: 37 with removed or broken dependencies, 88 with JavaScript errors (`SyntaxError`, `ReferenceError`, or `TypeError`). 269 packages had timeouts without instrumentation (e.g., waiting for user or network input). 258 packages had timeouts with our instrumentation due to large code size (including dependencies). In the remaining set of 9348 packages, NodeMedic detected 173 tainted flows.

Two security researchers independently manually review the 173 tainted flows to determine true and false positives. We define a true positive as a flow for which an exploit could be constructed and run successfully. Conversely, a false positive is a flow for which an exploit could not successfully be constructed. True positives could

TABLE 5. Large-scale evaluation breakdown

NodeMedic Result	# of Packages
Package Failure	125
Package Timeout	269
Instrumentation Timeout	258
No tainted flows	9175
Tainted flow	173
Total	10000

TABLE 6. Exploit synthesis results breakdown

Type	Count	Confirmed	Percent
ACI	133	102	77%
ACE	22	6	27%
Total	155	108	70%

be missed if neither researcher could construct an exploit, so our measure is a conservative lower-bound.

Through manual analysis of those flows we determine that 15 were false positives (8.7%) due to sanitization or flows to non-exploitable parameters of the sink (e.g., a flow to the "options" argument of `exec`), 3 are technically exploitable but we conservatively mark them as not vulnerable (e.g., the exploitable function is clearly marked as unsafe). This leaves 155 true vulnerabilities, of which 22 are ACE and 133 are ACI.

In Table 4, column 1 summarizes characteristics of all tainted flows; column 2, just of vulnerabilities. On average, vulnerable packages had 21K downloads in 2022. They had a mean of 6.31 dependencies with a tree depth of 1.12, and had on average 5.3K lines of code. Their provenance graphs were large, with 267 nodes and a depth of 36 nodes. Vulnerable packages executed 1.46 dependencies in the vulnerable execution trace, and sinks were typically found in the package itself. In general, complex vulnerabilities had larger provenance graphs and executed more dependencies. Detailed characteristics are in Appendix D.6, with case studies in Appendix D.5. In Appendix D.2 we show runtime per pipeline stage.

We performed responsible disclosure for the true vulnerabilities. At the time of publication, we have received four CVEs, and responses from 7 developers confirming vulnerabilities. We discuss our disclosure process in Appendix D.3 and developer response in Appendix D.4.

Result 5: NodeMedic scalably analyzes 10k packages, finding 173 flows (8.7% FP); 155 vulnerablities.

5.5.1. Exploit synthesis and confirmation. Next, we examine the effectiveness of our methodology for exploit synthesis and automatically confirming tainted flows. We present a summary of the results in Table 6. Overall, 70% of the true positives were automatically confirmed via our exploit synthesis technique. 77% of ACI vulnerabilities are automatically confirmed. On the other had, only 27% of ACE vulnerabilities could be automatically confirmed. This matches the intuition that ACE vulnerabilities, requiring the injection of a payload that is valid in the context of existing JavaScript code, are harder to exploit than ACI vulnerabilities, which utilize simpler shell code payloads.

Causes of exploit synthesis and confirmation failure. In order to understand limitations, we survey the vulnerabilities that failed to be automatically confirmed and categorize the reasons; presented in Table 7.

TABLE 7. Causes of synthesis and confirmation failure

Category	#ACE	#ACI	Total
Requires structured object	2	13	15
Driver insufficient	8	6	14
SMT unsatisfiable	2	10	12
Malformed payload	4	2	6
Total	16	31	47

The most common reason for failure is due to the package requiring a structured object to be supplied as an argument, with a property that contains the payload. This is beyond the capability of our exploit synthesis, which only supports the creation of exploits as strings.

The second most common issue is that our generated driver is insufficient. For example, a valid file path must be supplied or a callback function must be provided.

Finally, we have twelve cases of unsatisfiable SMT formulae and six cases where the synthesized exploit payload fails to be well-formed JavaScript or shell code, e.g., SMT-derived assignments to some inputs are not syntactically legal, indicating more constraints are needed.

Result 6: Automated exploit synthesis using provenance graphs confirms 70% of encountered vulnerabilities. It is most effective for ACI vulnerabilities (77%).

5.5.2. Triage model ratings. We measure safety (Definition 5.1) and precision (Definition 5.2) of the triage rating model on the (173) packages with tainted flows from our large-scale evaluation.

Safety of ratings. The model must avoid under-approximation of exploitability; it is undesirable for an analyst to deprioritize a flow that may be easily exploited.

Definition 5.1 (Triage Safety)**.** Given Provenance-AD-trees $\{t_{p_1}, \cdots, t_{p_n}\}$, a triage model R_M, and an expert R_E, we compute γ, the fraction of unsafe disagreements, $R_M(t_{p_i}) < R_E(t_{p_i})$. We say a model is *safe* if $\gamma \leq \gamma_0$.

We select $\gamma_0 = 0.05$ (5% unsafe), which is typically used to assign statistical significance [26].

Precision of ratings. Safety by itself would permit a trivial model that always assigns a "High" rating. Precision ensures triage model ratings are close to expert ratings.

Definition 5.2 (Triage Precision)**.** Given Provenance-AD-trees $\mathcal{H} = \{t_{p_1}, \cdots, t_{p_n}\}$, a triage model R_M, and an expert R_E, we compute $\hat{R_j} = \{R_j(t_{p_i}) \ \forall t_{p_i} \in \mathcal{H}\}$ for $j \in M, E$. We compute the agreement using an *inter-rater reliability* scoring function, $\kappa : (\hat{R_M}, \hat{R_E}) \rightarrow [0, 1]$. We say the model is *precise* if $\kappa \geq \kappa_0$.

Krippendorff's alpha [48] is κ; "substantial", "strong" agreement ($\kappa_0 = 0.61$) indicates precision [49], [60].

Expert ratings. We derive ratings from a panel of three experts with several combined years of experience uncovering, diagnosing, and repairing Node.js package vulnerabilities, in two steps: 1) Calibration; 2) Full rating.

During the calibration step we follow standard procedure for ensuring inter-rater reliability between the expert ratings [10], [30]. First, we select a random subset (20%) of the 173 tainted flows. Independently, all three raters provide a "High", "Medium", or "Low" exploitability rating for each package. Following independent rating of the 20% subset, the raters compare their ratings. We find that there is an "almost perfect" agreement amongst the three raters (0.97, measured with Krippendorff's alpha). The raters then resolve disagreements (only one).

Finally, the raters develop a shared rubric for evaluating package exploitability: "High": No particular input structure or configuration; lacks sanitization. "Medium": Structured input formats or implicit sanitization. "Low": Complex setup, comprehensive sanitization, or requires complex exploit payload construction. During the full rating step, ratings are assigned using the rubric by an expert to the remaining packages; in aggregate, 173 ratings.

Triage model ratings. We then apply the triage rating model to all 173 Provenance-Ad-trees. The model assigns a rating of "High" to 113 packages, "Medium" to 36 packages, and "Low" to 24 packages. The average time to compute a model rating is 0.07 seconds.

Triage safety. We find that there are just 4 unsafe disagreements between the expert ratings and the triage model ratings, amounting to 2.3% of all ratings. This meets our required threshold for Triage Safety (Definition 5.1), indicating that the triage model is safe. We examine the 4 packages (all of which are true positives) with unsafe disagreements to determine their causes and find that the general cause of unsafe model ratings is due to packages with long paths from attacker-controlled sources to vulnerable sinks.

Triage precision. Krippendorff's alpha, κ, is 0.74, indicating "substantial" agreement [49], [60] between expert and model ratings. This exceeds our threshold for triage precision ($\kappa_0 = 0.61$), indicating that the model is precise.

We explore two cases where there are disagreements (both *safe*): 1) Expert rates low-exploitability, model rates high-exploitability (9 packages; 5.2%). This discrepancy is because these flows reach non-exploitable sink arguments. The model can be improved with heuristics accounting for which sink argument is reached. 2) Expert rates low or medium, model rates medium or high-exploitability, respectively (12 packages; 6.9%). The cause is packages requiring a particular input format (e.g., certain object fields). The model can be improved with heuristics assigning weight to particular object fields.

Result 7: Provenance-graph–based triage rating has 2.3% unsafe disagreements and substantial agreement ($\kappa = 0.74$) with expert ratings.

6. Limitations and Future Work

Scope. NodeMedic focuses on two server-side dataflow vulnerabilities; ACE [18] and ACI [17]. Other prevalent dataflow vulnerabilities for Node.js packages are prototype pollution and hidden property attacks [42], [51], [84], [102]. As future work, NodeMedic can be extended to detect these via additional taint policies (Appendix A).

NodeMedic, like related analyses [41], [90], [91], does not consider implicit flows; related work demonstrates they provide little utility in detecting vulnerabilities in server-side JavaScript [92].

Like related tools [41], [91], NodeMedic relies on policies for precise analysis of native operations. This is a limitation of instrumentation-based dynamic analyses [32], [41], [76], [77]. NodeMedic provides policies for common native operations (Section 3.2.2).

While intended for offline use and faster than heavyweight analysis (e.g., [67]), NodeMedic does have 20x overhead, making it unsuitable for runtime monitoring. We leave provenance analysis optimization for future work.

Automation. Auto-tuning of propagation policies is coarse-grained (fully imprecise or precise). Fine-grained tuning requires manual work. As future work, auto-tuning can be extended to support fine-grained policy selection.

Discovering *all* vulnerable flows requires full coverage of a package. NodeMedic does not measure or adapt to coverage. This can be addressed by incorporating techniques such as fuzzing (e.g., SoFi [35]) or dynamic symbolic execution (e.g., ExpoSE [53]).

NodeMedic's generated driver may execute the package with semantically invalid input, and does not mock up external configuration (e.g., environment variables) or dependencies (e.g., a database). As future work, static analysis could be used to generate better drivers.

Just 27% of ACE vulnerabilities had successful exploits synthesized. This is due to the difficulty of synthesizing well-formed JavaScript objects and code that satisfy constraints implied by the provenance graph, which we leave as future work.

Threats to validity. The triage model is evaluated against experts because Node.js package exploitability lacks quantitative measurement. We ensured agreement amongst raters with a consistent rubric (Section 5.5.2), but had a small sample of ratings. As future work, larger studies should be conducted.

7. Related Work

We first discuss work on the general security issues of Node.js. Then, we cover dynamic taint analysis [81] for JavaScript (for a survey, see [1]), focusing on tools similar to NodeMedic, then discussing its formal foundations, and concluding with JavaScript taint analyses in other settings. Finally, we discuss connections to existing end-to-end infrastructures, analyses conceptually similar to provenance, and tools generating JavaScript exploits.

Node.js platform and ecosystem security. Several studies measure Node.js ecosystem security [25], [33], [92], [103], [105] and find serious architectural issues with Node.js [74] and issues in its packages [92]. Node.js lacks mechanisms for sandboxing or moderating third-party packages; to remedy these issues, changes to the Node.js architecture have been proposed [22], [96], but lack wide adoption.

In contrast, many analyses improve security without architectural changes [42]–[44], [52], [53], [55], [68], [78], [79], [90], [91], [96], [101], [102]; we compare work related to NodeMedic's provenance analysis below.

Taint analysis of Node.js packages. Several tools, including NodeMedic, perform taint analysis of Node.js packages [32], [41], [51], [62], [67], [79], [90]. The closest to NodeMedic is Ichnaea [41] which also uses Jalangi2 [82], [83] for dynamic taint analysis. Unlike NodeMedic, Ichnaea does not track provenance, instead tracking boolean taint and providing minimal feedback (i.e., existence of a flow). Ichnaea utilizes a separate abstract stack-based machine to propagate taint and is less precise than NODEMEDIC likely due to the difficulty of replicating JavaScript semantics in the abstract machine.

Affogato [32] also implements dynamic taint analysis, but performs taint propagation for string operations via an inference mechanism based on string similarity; it does not support precise tainting of strings like NODEMEDIC.

Nodest [67], unlike NODEMEDIC, performs static taint analysis and uses abstract interpretation. Nodest achieves scalability by not precisely analyzing every dependency. This is similar to NODEMEDIC's approach of automatically tuning propagation policies.

JavaScript information flow monitoring techniques. Multiple works formalize foundational techniques for JavaScript information flow analysis [3], [13]–[15], [36], [40], [80]. Chudnov et al. proposed taint tracking by boxing primitive values [14]; this is used by other works [4], [24], [95]. Like NODEMEDIC, Chudnov et al. perform source-to-source rewriting of JavaScript programs. Unlike Chudnov et al., NODEMEDIC tracks provenance and does not store taint as properties on boxed values.

JavaScript taint analysis in other settings. In the client-side web setting, multiple works perform taint analysis [50], [61], [76], [77], [94], [97], [99]. DexterJS [76], [77] has an analysis methodology similar to NODEMEDIC. Like NODEMEDIC, they perform precise (character-level) tainting of strings. Unlike NODEMEDIC, they store taint information as properties on boxed values. For browser extensions analysis, some works modify the JavaScript engine to perform taint analysis [12], [23].

New JavaScript interpreters for taint tracking have also been created [7], [36], [37], [46], [47]. JSFlow [37], like NODEMEDIC, uses modeling to support precise analysis. However, unlike NODEMEDIC, JSFlow does not support character-level tainting of primitive strings, nor Node.js's built-in APIs. Kreindl et al. [46], [47] build a tool that performs taint analysis for multiple languages. While it is applicable to JavaScript analysis, it lacks support for Node.js-specific functionality.

End-to-end analysis infrastructure. Several client-side dynamic taint analyses are integrated into end-to-end infrastructures supporting automated discovery and confirmation of tainted flows [50], [61], [76]. In contrast, existing Node.js dynamic taint analyses require manual driver creation and exploit confirmation [32], [41].

The closest end-to-end infrastructure analyzing Node.js packages is MalOSS [98], Unlike NODEMEDIC, MalOSS's integrated taint analyzer, JSPrime [79], is static and imprecise, but does not require a driver. Additionally, MalOSS's vulnerability confirmation is semi-automated; heuristics flag packages for manual review. NODEMEDIC can synthesize and test exploits automatically.

Triage and provenance. Measuring exploitability has been discussed in other contexts. Newsome et al. measure *influence* with a precise static analysis that quantifies the set of feasible values for attacker-controllable data in x86 binaries [66]. Masri et al. quantify Java bytecode dataflow *strength* by measuring flow characteristics such as data dependence and length [59]. The data structures used in their work for triage contain information about dataflow, similar to NODEMEDIC's provenance graph. However, neither capture all operations performed on tainted data like NODEMEDIC. Consequently, we can use the provenance graph for both triage and exploit synthesis.

Data provenance is used for debugging database and logic programming query systems [8], [9], [104], where one may want to know all of the inputs that affect the outcome of a query. Similarly, the analysis techniques of dynamic dependence analysis and program slicing can extract paths of program points responsible for an issue [28], [56]–[58], [85]. Finally, works detecting Advanced and Persistent Threats (ATPs) perform kernel-level logging of system events to produce a causal graph that explains a threat [38], [39], [54]. While these are conceptually similar, they do not share the provenance graph structure, nor notions of triage rating or exploit synthesis.

JavaScript exploit generation. To the best of our knowledge NODEMEDIC is the first to synthesize exploits from taint provenance data based on package inputs, string constants, and operations. Most prior work targets cross-site scripting vulnerabilities [6], [27], [31], [50], [77] and parses the AST of the statement reaching the sink to construct an exploit [6], [50], [77]. This works in the web setting because global input sources (e.g., URL) are accessed near the time of sink execution, but cannot be directly applied to Node.js packages, where inputs are not global and can be processed well before sink execution.

PMForce [93] synthesizes ACE exploits for the client-side `postMessage` API's `event` object. Unlike NODEMEDIC, forced execution is performed, where particular branches are forced to be taken with proxied objects to gather path constraints. NODEMEDIC handles input types that cannot easily be proxied, and gathers concrete runtime traces; not path constraints. PMForce uses constraints to fill exploit templates to use for `event.data` field values. NODEMEDIC also uses templates, but these encode ACE or ACI-specific breakouts. The provenance graph encodes constraints that solve for the construction of inputs that ensure the exploit payload reaches the sink.

Lynx [102] targets hidden property abuse in Node.js packages and uses symbolic execution to find sinks influenced by hidden properties. If found, the symbolic hidden property is replaced by an indicator that signals exploitability. As future work, NODEMEDIC could be extended to support symbolic values in generated exploits to allow for symbolic execution.

8. Conclusion

We present NODEMEDIC, an end-to-end dynamic taint *provenance* analysis infrastructure for detecting ACE and ACI vulnerabilities in Node.js packages. NODEMEDIC's analysis generates *provenance graphs* that contain valuable information about how attacker-controllable data reaches a sink. NODEMEDIC reduces analysts' manual burden with automated package driver creation, and post-detection analyses that use provenance graphs: synthesizing candidate exploits for automated confirmation and Attack-defense-tree–based rating of flow exploitability. Our large-scale evaluation of 10,000 npm packages shows that NODEMEDIC is effective at detecting and confirming vulnerabilities in real Node.js packages.

Data Availability

NodeMedic is released as open-source software, including setup scripts and documentation required to reproduce the large-scale analysis presented in this paper, along with a suite of 589 taint precision unit tests. Available here: https://github.com/NodeMedicAnalysis/.

Acknowledgements

This work is supported in part by Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) through the Carnegie Mellon Portugal Program under the project DIVINA (CMU/TIC/0053/2021), Future Enterprise Security Initiative at Carnegie Mellon CyLab (FutureEnterprise@CyLab), and Carnegie Mellon CyLab.

References

[1] Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. A Survey of Dynamic Analysis and Test Generation for JavaScript. *ACM Computing Surveys*, 2017.

[2] Zaruhi Aslanyan, Flemming Nielson, and David Parker. Quantitative verification and synthesis of attack-defence scenarios. In *IEEE Computer Security Foundations Symposium (CSF)*, 2016.

[3] Thomas H. Austin, T. Disney, C. Flanagan, and A. Jeffrey. Dynamic Information Flow Analysis for Featherweight JavaScript. Technical Report UCSC-SOE-11-19, University of California Santa Cruz, 2011.

[4] Thomas H. Austin, Tim Disney, and Cormac Flanagan. Virtual values for language extension. In *ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2011.

[5] Babel. Babel - The compiler for next generation JavaScript, 2023. https://babeljs.io/.

[6] Souphiane Bensalim, David Klein, Thomas Barber, and Martin Johns. Talking about my generation: Targeted DOM-based XSS exploit generation using dynamic data flow analysis. In *European Workshop on Systems Security*, 2021.

[7] Abhishek Bichhawat, Vineet Rajani, Deepak Garg, and Christian Hammer. Information Flow Control in WebKit's JavaScript Bytecode. In *Principles of Security and Trust*, 2014.

[8] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. Why and where: A characterization of data provenance. In *International Conference on Database Theory (ICDT)*, 2001.

[9] Peter Buneman and Wang-Chiew Tan. Provenance in databases. In *ACM Special Interest Group on Management of Data*, 2007.

[10] Gemma Catolino, Fabio Palomba, Andy Zaidman, and Filomena Ferrucci. Not all bugs are the same: Understanding, characterizing, and classifying bug types. *Journal of Systems and Software*, 2019.

[11] CERT. The CERT guide to coordinated vulnerability disclosure, 2023. https://vuls.cert.org/confluence/display/CVD.

[12] Quan Chen and Alexandros Kapravelos. Mystique: Uncovering Information Leakage from Browser Extensions. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[13] A. Chudnov and D. A. Naumann. Information Flow Monitor Inlining. In *IEEE Computer Security Foundations Symposium (CSF)*, 2010.

[14] Andrey Chudnov and David A. Naumann. Inlined Information Flow Monitoring for JavaScript. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[15] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for javascript. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[16] The Unicode Consortium. FAQ - Private-Use Characters, Noncharacters, and Sentinels, 2023. https://www.unicode.org/faq/private_use.html.

[17] The MITRE Corporation. CWE - CWE-77: Improper Neutralization of Special Elements used in a Command ('Command Injection') (4.3), 2023. https://cwe.mitre.org/data/definitions/77.html.

[18] The MITRE Corporation. CWE - CWE-94: Improper Control of Generation of Code ('Code Injection') (4.3), 2023. https://cwe.mitre.org/data/definitions/94.html.

[19] The MITRE Corporation. CWE-22: Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal'), 2023. https://cwe.mitre.org/data/definitions/22.html.

[20] CVE. CVE organization homepage. https://www.cve.org/.

[21] National Vulnerability Database. National vulnerability database vulnerability metrics. https://nvd.nist.gov/vuln-metrics/cvss.

[22] Willem De Groef, Fabio Massacci, and Frank Piessens. NodeSentry: Least-privilege library integration for server-side JavaScript. In *Annual Computer Security Applications Conference*, 2014.

[23] M. Dhawan and V. Ganapathy. Analyzing Information Flow in JavaScript-Based Browser Extensions. In *Annual Computer Security Applications Conference*, 2009.

[24] Tim Disney, Nathan Faubion, David Herman, and Cormac Flanagan. Sweeten your JavaScript: Hygienic macros for ES5. *ACM SIGPLAN Notices*, 2015.

[25] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. Towards measuring supply chain attacks on package managers for interpreted languages. In *Network and Distributed System Security (NDSS)*, 2021.

[26] R. A. Fisher. Statistical methods for research workers. In *Breakthroughs in Statistics: Methodology and Distribution*, 1992.

[27] Yaw Frempong., Yates Snyder., Erfan Al-Hossami., Meera Sridhar., and Samira Shaikh. HIJaX: Human intent javascript XSS generator. In *International Conference on Security and Cryptography (SECRYPT)*, 2021.

[28] Xiaoqin Fu and Haipeng Cai. FlowDist: Multi-Staged Refinement-Based dynamic information flow analysis for distributed software systems. In *USENIX Security Symposium (USENIX)*, 2021.

[29] Olga Gadyatskaya, René Rydhof Hansen, Kim Guldstrand Larsen, Axel Legay, Mads Chr. Olesen, and Danny Bøgsted Poulsen. Modelling attack-defense trees using timed automata. In *Formal Modeling and Analysis of Timed Systems*, 2016.

[30] Zheng Gao, Christian Bird, and Earl T. Barr. To type or not to type: Quantifying detectable bugs in JavaScript. In *International Conference on Software Engineering (ICSE)*, 2017.

[31] Behrad Garmany, Martin Stoffel, Robert Gawlik, Philipp Koppe, Tim Blazytko, and Thorsten Holz. Towards automated generation of exploitation primitives for web browsers. In *Annual Computer Security Applications Conference*, 2018.

[32] François Gauthier, Behnaz Hassanshahi, and Alexander Jordan. AFFOGATO: Runtime detection of injection attacks for Node.js. In *ISSTA/ECOOP Workshops*, 2018.

[33] Liang Gong. *Dynamic Analysis for JavaScript*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2018.

[34] René Hansen, Peter Jensen, Kim Larsen, Axel Legay, and Danny Poulsen. Quantitative evaluation of attack defense trees using stochastic timed automata. In *Graphical Models for Security*, 2018.

[35] Xiaoyu He, Xiaofei Xie, Yuekang Li, Jianwen Sun, Feng Li, Wei Zou, Yang Liu, Lei Yu, Jianhua Zhou, Wenchang Shi, and Wei Huo. SoFi: Reflection-augmented fuzzing for javascript engines. In *ACM Conference on Computer and Communications Security (CCS)*, 2021.

[36] D. Hedin and A. Sabelfeld. Information-Flow Security for a Core of JavaScript. In *IEEE Computer Security Foundations Symposium (CSF)*, 2012.

[37] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. JSFlow: Tracking information flow in JavaScript and its APIs. In *Annual ACM Symposium on Applied Computing (SAC)*, 2014.

[38] Md Nahid Hossain, Sadegh M. Milajerdi, Junao Wang, Birhanu Eshete, Rigel Gjomemo, R. Sekar, Scott Stoller, and V. N. Venkatakrishnan. SLEUTH: Real-time attack scenario reconstruction from COTS audit data. In *USENIX Security Symposium (USENIX)*, 2017.

[39] Md Nahid Hossain, Sanaz Sheikhi, and R. Sekar. Combating dependence explosion in forensic analysis using alternative tag propagation semantics. In *IEEE Symposium on Security and Privacy (SP)*, 2020.

[40] Seth Just, Alan Cleary, Brandon Shirley, and Christian Hammer. Information flow analysis for JavaScript. In *ACM SIGPLAN International Workshop on Programming Language and Systems Technologies for Internet Clients*, 2011.

[41] R. Karim, F. Tip, A. Sochurkova, and K. Sen. Platform-Independent Dynamic Taint Analysis for JavaScript. *IEEE Transactions on Software Engineering (TSE)*, 2018.

[42] Hee Yeon Kim, Ji Hoon Kim, Ho Kyun Oh, Beom Jin Lee, Si Woo Mun, Jeong Hoon Shin, and Kyounggon Kim. DAPP: automatic detection and analysis of prototype pollution vulnerability in Node.js modules. *International Journal of Information Security*, 2021.

[43] Maryna Kluban, Mohammad Mannan, and Amr Youssef. On measuring vulnerable javascript functions in the wild. In *ACM ASIA Conference on Computer and Communications Security (ASIACCS)*, 2022.

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

[45] Barbara Kordy, Sjouke Mauw, Saša Radomirović, and Patrick Schweitzer. Foundations of attack–defense trees. In *Formal Aspects of Security and Trust*, 2011.

[46] Jacob Kreindl, Daniele Bonetta, and Hanspeter Mössenböck. Towards efficient, multi-language dynamic taint analysis. In *ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, 2019.

[47] Jacob Kreindl, Daniele Bonetta, Lukas Stadler, David Leopoldseder, and Hanspeter Mössenböck. Multi-language dynamic taint analysis in a polyglot virtual machine. In *ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, 2020.

[48] Klaus Krippendorff. Computing krippendorff's alpha-reliability. *Departmental Papers (UPenn ASC)*, 2011.

[49] J. Richard Landis and Gary G. Koch. The measurement of observer agreement for categorical data. *Biometrics*, 1977.

[50] Sebastian Lekies, Ben Stock, and Martin Johns. 25 million flows later: Large-scale detection of DOM-based XSS. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[51] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi 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*, 2021.

[52] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. Mining node.js vulnerabilities via object dependence graph and query. In *USENIX Security Symposium (USENIX)*, 2022.

[53] Blake Loring, Duncan Mitchell, and Johannes Kinder. ExpoSE: Practical symbolic execution of standalone JavaScript. In *ACM SIGSOFT International SPIN Symposium on Model Checking of Software (SPIN)*, 2017.

[54] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. ProTracer: Towards practical provenance tracing by alternating between logging and tainting. In *Network and Distributed System Security (NDSS)*, 2016.

[55] Magnus Madsen, Frank Tip, and Ondřej Lhoták. Static analysis of event-driven Node.js JavaScript applications. *ACM SIGPLAN Notices*, 2015.

[56] W. Masri, A. Podgurski, and D. Leon. Detecting and debugging insecure information flows. In *International Symposium on Software Reliability Engineering*, 2004.

[57] Wes Masri and Andy Podgurski. Application-based anomaly intrusion detection with dynamic information flow analysis. *Computer Security*, 2008.

[58] Wes Masri and Andy Podgurski. Algorithms and tool support for dynamic information flow analysis. *Information and Software Technology*, 2009.

[59] Wes Masri and Andy Podgurski. Measuring the strength of information flows in programs. *ACM Transactions on Software Engineering Methodolology*, 2009.

[60] Gareth McCray. Assessing inter-rater agreement for nominal judgement variables. In *Language Testing Forum*, 2013.

[61] William Melicher, A. Das, Mahmood Sharif, L. Bauer, and Limin Jia. Riding out DOMsday: Towards Detecting and Preventing DOM Cross-Site Scripting. In *Network and Distributed System Security (NDSS)*, 2018.

[62] Jiang Ming, Dinghao Wu, Gaoyao Xiao, Jun Wang, and Peng Liu. TaintPipe: Pipelined symbolic taint analysis. In *USENIX Security Symposium (USENIX)*, 2015.

[63] Mozilla. JavaScript eval function documentation, 2023. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval.

[64] Mozilla. JavaScript Function constructor documentation, 2023. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function.

[65] Phil Muncaster. Open Source Supply Chain Attacks Surge 430%, 2020. https://www.infosecurity-magazine.com/news/open-source-supply-chain-attacks/.

[66] James Newsome, Stephen McCamant, and Dawn Song. Measuring channel capacity to distinguish undue influence. In *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, 2009.

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

[68] Benjamin Barslev Nielsen, Martin Toldam Torp, and Anders Møller. Modular call graph construction for security scanning of node.js applications. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021.

[69] Node.js. Node.js exec API documentation, 2023. https://nodejs.org/api/child_process.html#child_processexeccommand-options-callback.

[70] Node.js. Node.js execFile API documentation, 2023. https://nodejs.org/api/child_process.html#child_processexecfilefile-args-options-callback.

[71] Node.js. Node.js execSync API documentation, 2023. https://nodejs.org/api/child_process.html#child_processexecsynccommand-options.

[72] npm. npm deprecate command, 2023. https://docs.npmjs.com/cli/v9/commands/npm-deprecate.

[73] npm. npm unpublish command, 2023. https://docs.npmjs.com/cli/v9/commands/npm-unpublish.

[74] A. Ojamaa and K. Düüna. Assessing the security of Node.js platform. In *International Conference for Internet Technology and Secured Transactions*, 2012.

[75] Stack Overflow. Stack Overflow Developer Survey, 2022. https://survey.stackoverflow.co/2022/.

[76] Inian Parameshwaran, Enrico Budianto, Shweta Shinde, Hung Dang, Atul Sadhu, and Prateek Saxena. Auto-patching DOM-based XSS at scale. In *Joint Meeting on Foundations of Software Engineering*, 2015.

[77] Inian Parameshwaran, Enrico Budianto, Shweta Shinde, Hung Dang, Atul Sadhu, and Prateek Saxena. DexterJS: Robust testing platform for DOM-based XSS vulnerabilities. In *Joint Meeting on Foundations of Software Engineering*, 2015.

[78] Joonyoung Park, Jihyeok Park, Dongjun Youn, and Sukyoung Ryu. Accelerating javascript static analysis via dynamic shortcuts. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021.

[79] Nishant Patnaik and Sarathi Sahoo. Javascript static security analysis made easy with JSPrime. In *Blackhat USA*, 2013.

[80] José Fragoso Santos and Tamara Rezk. An Information Flow Monitor-Inlining Compiler for Securing a Core of JavaScript. In *ICT Systems Security and Privacy Protection*, 2014.

[81] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE Symposium on Security and Privacy (SP)*, 2010.

[82] Koushik Sen. Jalangi2 repository. https://github.com/Samsung/jalangi2.

[83] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In *Joint Meeting on Foundations of Software Engineering*, 2013.

[84] Mikhail Shcherbakov, Musard Balliu, and Cristian-Alexandru Staicu. Silent spring: Prototype pollution leads to remote code execution in Node.js. In *USENIX Security Symposium (USENIX)*, 2023.

[85] Paritosh Shroff, Scott Smith, and Mark Thober. Dynamic dependency monitoring to secure information flow. In *IEEE Computer Security Foundations Symposium (CSF)*, 2007.

[86] Snyk. Disclose a vulnerability in an open source package. https://docs.snyk.io/more-info/disclosing-vulnerabilities/disclose-a-vulnerability-in-an-open-source-package.

[87] Snyk. Snyk homepage. https://snyk.io/.

[88] Snyk and EffectRenan. Command injection vulnerability in systeminformation, 2020. https://www.npmjs.com/advisories/1590.

[89] Snyk and NodeMedic. CVE and CVSS for *font-converter*, 2022. https://security.snyk.io/vuln/SNYK-JS-FONTCONVERTER-2976194.

[90] C.-A. Staicu, M. T. Torp, M. Schäfer, A. Møller, and M. Pradel. Extracting Taint Specifications for JavaScript Libraries. In *International Conference on Software Engineering (ICSE)*, 2020.

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

[92] Cristian-Alexandru Staicu, Daniel Schoepe, Musard Balliu, Michael Pradel, and Andrei Sabelfeld. An Empirical Study of Information Flows in Real-World JavaScript. In *ACM SIGSAC Workshop on Programming Languages and Analysis for Security*, 2019.

[93] Marius Steffens and Ben Stock. PMForce: Systematically analyzing postMessage handlers at scale. In *ACM Conference on Computer and Communications Security (CCS)*, 2020.

[94] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. TAJ: Effective taint analysis of web applications. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[95] Tom Van Cutsem and Mark S. Miller. Proxies: Design principles for robust object-oriented intercession APIs. *ACM SIGPLAN Notices*, 2021.

[96] Nikos Vasilakis, Cristian-Alexandru Staicu, Grigoris Ntousakis, Konstantinos Kallas, Ben Karel, André DeHon, and Michael Pradel. Preventing dynamic library compromise on node.js via RWX-based privilege reduction. In *ACM Conference on Computer and Communications Security (CCS)*, 2021.

[97] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Krügel, and Giovanni Vigna. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Network and Distributed System Security (NDSS)*, 2007.

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

[99] G. Wassermann and Z. Su. Static detection of cross-site scripting vulnerabilities. In *International Conference on Software Engineering*, 2008.

[100] Wojciech Wideł. *Formal modeling and quantitative analysis of security using attack-defense trees*. PhD thesis, INSA de Rennes, 2019.

[101] Elizabeth Wyss, Alexander Wittman, Drew Davidson, and Lorenzo De Carli. Wolf at the door: Preventing install-time attacks in npm with latch. In *ACM ASIA Conference on Computer and Communications Security (ASIACCS)*, 2022.

[102] Feng Xiao, Jianwei Huang, Yichang Xiong, Guangliang Yang, Hong Hu, Guofei Gu, and Wenke Lee. Abusing hidden properties to attack the node.js ecosystem. In *USENIX Security Symposium (USENIX)*, 2021.

[103] N. Zahan, T. Zimmermann, P. Godefroid, B. Murphy, C. Maddila, and L. Williams. What are weak links in the npm supply chain? In *International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2022.

[104] David Zhao, Pavle Subotić, and Bernhard Scholz. Debugging large-scale datalog: A scalable provenance evaluation strategy. *ACM Transactions on Programming Language and Systems*, 2020.

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

A. Precise Propagation Policies

String module policies. For strings, in addition to the precise string encoding, precise policies are implemented for common string operations like `substring`, `slice`, `split`, `charCodeAt`, `fromCharCode`. Implementation of policies can be very simple using our framework. For example, the precise taint policy for `toLowerCase` is written in just 10 lines of code, not counting boilerplate for accessing taint information.

The most complex string taint policy is that of `split`. The difficulty stems from the fact that `split` is a *cross-type policy*; `split` converts a string into an array, and the precision level of the string and array policies may not be the same. For example, if strings are precisely tainted while arrays are imprecisely tainted then the resulting array needs to be tainted if *any* character of the original string was tainted. However, if both are using precise policies then the taint of characters need to correspond one-to-one with the taint of the resulting array elements that contain those characters. NODEMEDIC enables a complex policy like this to be written in about 100 lines of code.

Array module policies. *Imprecise*: Each array element shares the provenance node of the array. *Precise*: Every element has its own provenance node, allowing for precise propagation for many for array functions and operations without additional policies. Policies are needed

```
// Policy for setting a field
TPutField(s, v1, v2, v3) {
// Check if v3 is tainted
// u2 is unwrapped v2
if (v3tainted && !isUndefinedOrNull(u2)) {
  let pollution = false;
  // Case 1: Overwriting prototype chain
  let pr = ['prototype', '__proto__'];
  if (pr.indexOf(u2.toString()) != -1) {
      pollution = true; }
  // Case 2: Setting a prototype property
  if (u1 == Object.prototype) {
      pollution = true; }
  ... }}
```

Figure 15. Example object policy for prototype pollution.

for the higher-order functions, e.g., `join`, `map`, `reduce`, `reduceRight`. As pointed out by prior work, precise propagation can be difficult for these because of the potential interleaving of native and non-native callbacks [41].

Our framework makes handling these higher-order functions simpler. The policy for `reduce` is implemented in around 30 lines of code and is precise. The reason is due to our underlying representation of precise array tainting; since individual array elements have their only entries in the taint map, M_t, propagation can be handled by our regular provenance propagation semantics. This holds except in the case where an *uninstrumented* callback is passed to `reduce`. In this case, the `reduce` policy steps in to imprecisely propagate provenance (since the behavior of the function body is unknown to the instrumentation).

Object and global module policies. For objects, precise function policies are not necessary except for the `defineProperty` function. This function defines a new property on an object. It accepts a `PropertyDescriptor` object that contains the value of the new property. The fields of the `PropertyDescriptor` must not be wrapped in order for setting the property to work correctly. In this case we perform additional bookkeeping to ensure that the provenance of `PropertyDescriptor` is preserved and the new property is defined correctly.

Policies for PRP and HPA. NodeMedic supports the addition of new policy definitions, scoped to particular objects and modules. Below, we describe what would be needed to add support for policies for detecting prototype pollution (PRP) and hidden property abuse (HPA). For PRP, one would have to add new policy code to the base `Object` policy that would flag a write with tainted values to the prototype (Figure 15). For HPA, following a similar code pattern, policies could be added for functions such as `Object.assign` to check whether tainted values are being assigned to known-sensitive fields. The set of known-sensitive fields would have to be separately user-annotated or determined via a pre-analysis.

Finally, as future work, additional investigation is still needed to develop methodology that enables triage and exploit synthesis for these vulnerabilities, and additional engineering effort would be required to implement this in NodeMedic's end-to-end infrastructure.

B. String Taint Encoding

The unicode code point range has hexidecimal values that are designated as "private use" [16]; this range can be used without concern for collision with other unicode code-points. NodeMedic performs a transformation $SEnc : S \times T \rightarrow S'$ from a string, S, and an array indicating which characters of the string are tainted, T, to a new string, S', that includes the taint bits in the unicode private use area. The encoded string can then be used for native taint propagation in JavaScript operations that transform strings: `str` → `str`, including string concatenation, slicing, substring, and indexing.

Operations that take a string and produce a non-string data type (e.g., `charCodeAt`) do not retain their semantics with this encoding. Operations that attempt to perform character-level comparisons also produce incorrect results without encoding the compared string as well; for example, `str.indexOf(x)`, unless `x` is also encoded.

C. Atk Automata Construction

Below we present details on the construction of Atk automata for triage rating.

Definition C.1 (Probabilistic Attacker Automata)**.** Let Atk be a five-tuple (Q, E, δ, F, P) where Q is a set of finite states in $2^{\bar{\mathcal{V}}}$. E is a finite set of input symbols corresponding to the elements of A_d. δ is a transition function $Q \times E \rightarrow Q$ that assigns edges according to t_p. F is the set of final states; success or failure: $\{q_s, q_f\}$. P is a function $Q \times E \rightarrow [0, 1]$ assigning the probability of transition from $q \xrightarrow{a} q'$ for each $a \in E$, under the constraint that $\sum_{q'} Pr[q \xrightarrow{a} q'] = 1$.

As described in Section 4.3, we represent the attacker's input via families of strings, $\mathcal{V}_1, \ldots, \mathcal{V}_n$, under the assumption that different implicit and explicit sanitizations will cover different exploits. The behavior for an attacker during one execution of the program is to try many exploits. For example, an attacker with control over two independent inputs v_1, v_2 of f_i that both reach a sink should provide a *different* exploit for v_1 and v_2 during one run of f_i. The set of attacker's exploits is thus $\bar{\mathcal{V}} = \{\mathcal{V}_1, \ldots, \mathcal{V}_k\}$ where k is stochastically-sampled, proportional to the number of attacker-controllable inputs.

P uses the constant ϵ, which characterizes the average Provenance-AD-tree. Given a population of trees it is inversely proportional to the average path length, D, and average number of nodes in attacker-controlled paths, μ: $\epsilon = \frac{1}{\mu * D}$. P is defined over failure of defender actions: $\mathcal{O}_I$: $\frac{|\bar{\mathcal{V}}|-1}{|\bar{\mathcal{V}}|}$; At least one $\mathcal{V}_i \in \bar{\mathcal{V}}$ succeeds. $\mathcal{O}_E$: $\frac{1}{|\bar{\mathcal{V}}|}$; At most one $\mathcal{V}_i \in \bar{\mathcal{V}}$ succeeds. $\mathcal{O}_A$: $1 - \epsilon|\bar{\mathcal{V}}|$; The probability is proportional to $|\bar{\mathcal{V}}|$, and scaled by ϵ. $\mathcal{O}_\star$: $1 - \epsilon$; The probability is constant.

Next, we must instantiate each of (Q, E, δ, F, P), given a particular t_p. We call this process *compiling* Atk:

1) Q is drawn from $2^{\bar{\mathcal{V}}}$. In practice the ordering of the set $\{\mathcal{V}_1, \cdots, \mathcal{V}_k\}$ does not matter so we have a group (of unique sets) of size $|\bar{\mathcal{V}}|$, i.e., $Q = \{q_1, q_2, \ldots, q_{|\bar{\mathcal{V}}|}\}$.

2) E is the set of $o \in t_p$ $(o \in \mathcal{O})$.

3) δ assigns an edge $q_i \xrightarrow{a} q_j$ for $q_i, q_j \in |\bar{\mathcal{V}}|$ according to the categories of $a \in E$:

- $a \in \mathcal{O}_I$: $q_i \xrightarrow{a} q_{i-1}$ for success; $q_i \xrightarrow{a} q_f$ for failure.
- $a \in \mathcal{O}_E$: $q_i \xrightarrow{a} q_1$ for success; $q_i \xrightarrow{a} q_f$ for failure.
- $a \in \mathcal{O}_A \cup \mathcal{O}_\star$: $q_i \xrightarrow{a} q_i$ for success; $q_i \xrightarrow{a} q_f$ for failure.

Algorithm 2 Bernoulli Estimation of $Pr[E_{S_{\text{ATK}}}]$

```
ATK ← COMPILE($t_p, \epsilon$)
success ← 0, failure ← 0, $i$ ← 0
while $i$ < SIMULATIONS do
    while ATK.$q \notin \{q_s, q_f\}$ do
        $p_\eta$ ← NEXT($t_p, p_\eta$)
        ATK.$q$ ← ATK($p_\eta.p$)
    end while
    if ATK.$q = q_s$ then
        success ← success + 1
    else if ATK.$q = q_f$ then
        failure ← failure + 1
    end if
    $i$ ← $i + 1$
end while
return $\frac{\text{success}}{\text{success + failure}}$
```

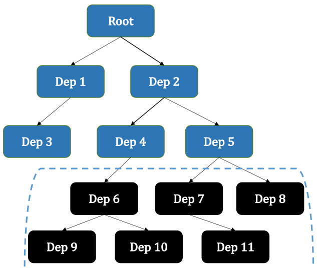


Figure 16. Ex: Algorithm 1 with $\mathtt{m_{deps}} = 0$ and $\mathtt{m_{depth}} = 3$. Functions of dependencies below the dashed line are over-approximately analyzed.

- $a \in \mathcal{O}_S$: $q_i \xrightarrow{a} q_s$ if $a \in SN_1$ and $|\bar{\mathcal{V}}| \geq 1$ or $a \in SN_2$ and $|\bar{\mathcal{V}}| > 1$ otherwise $q_i \xrightarrow{a} q_f$.

The intuition is that an implicit sanitization $\mathcal{O}_I$ is not comprehensive; we model it as only successful at eliminating one of ATK's payloads, while an explicit sanitization $\mathcal{O}_E$ will render ineffective all-but-one of the ATK's payloads. Other nodes do not cause reduction of payloads, but can still transition to a failure state (e.g., if the program halts at that point). Finally, we model success and failure for separately ACI and ACE: $\{\mathtt{exec}, \mathtt{execSync}\} \in SN_1, \{\mathtt{eval}, \mathtt{Function}\} \in SN_2$; ACI sinks are typically easier to exploit than ACE (Section 5.5.1).

4) F is the set of two final states: q_s representing a successful attack, i.e., $f_i(v_{\text{ATK}})$ succeeded; q_f representing the failure state, i.e., $f_i(v_{\text{ATK}})$ failed.

5) Finally, P is as described previously.

The compiled automata can then be used for estimation of $Pr[E_{S_{\text{ATK}}}]$. We presented the methodology in Section 4.3 but include pseudocode here: Algorithm 2.

D. Additional Evaluation Details

D.1. Additional Results for Prior Vulnerabilities

In Table 8 we present additional results for the prior vulnerabilities dataset, namely triage and exploit synthesis. Note that only true positives are included in this table because true negatives do not produce provenance graphs for these packages (NODEMEDIC has no false positives for this set, as previously discussed in Section 5.2). Only packages with generated provenance graphs can be used with our triage and exploit synthesis techniques.

Since we already have provenance graphs for these packages, we do not run them with the full end-to-end pipeline, rather, we only programmatically run triage and exploit synthesis for them. To run our triage model, we invoke the model with the provenance graph for each package and record the result (as described in Section 4.3). To run exploit synthesis on these packages, we pass the previously generated provenance graph to our SMT formula generator. As described in Section 4.2, we solve the generated formula with Z3, extract the solved strings (if sat), and rerun the existing driver, substituting the solved strings for the tainted inputs, and check to see if the generated exploit was successful.

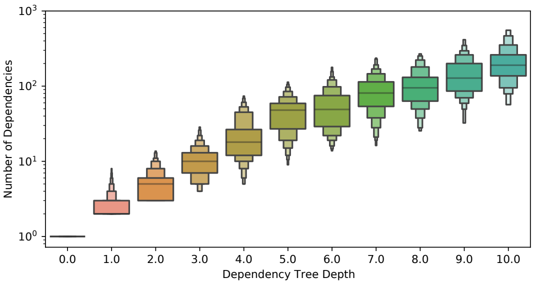


Figure 17. Number of dependencies and depth of dependency trees of packages from large-scale evaluation.

TABLE 8. ANALYSIS OF PACKAGES FROM PRIOR WORK

Package	Auto-conf.	Rating
fish	✓	H
git2json	✓	H
gm	✓	H
growl	✓	H
kerb_request	✓	H
m-log	✓	M
mixin-pro	-	L
mobile-icon-resizer	-	H
mol-proto	-	H
mongo-parse	-	M
mongoosemask	-	M
mongoosify	-	M
node-libnotify	✓	H
node-os-utils	✓	H
office-converter	✓	H
pidusage	✓	H
pomelo-monitor	✓	H
systeminformation	✓	H

D.2. End-to-End Analysis Runtime Breakdown

Our large-scale evaluation completed in four days of parallelized execution across 16 Docker containers, each hosting an instance of our end-to-end pipeline. The average (per package) runtime in seconds per end-to-end pipeline stage is listed in Table 9. “Package setup” indicates the time spent downloading and installing the package from npm. “Driver generation” is the time spent pre-analyzing the package to gather its public functions and generating driver code (Section 4.1). “Precision tuning” includes the time spend building the package dependency tree and applying the auto-tuning algorithm to it (Section 3.3). “Execution” is the total time spent

TABLE 9. AVERAGE RUNTIME PER PIPELINE STAGE

Stage	Runtime (s)
Package setup	1.24
Driver generation	0.24
Precision tuning	19.64
Execution	60.88
Triage rating	0.07
Exploit synthesis	32.88

executing the package, including time spent analyzing every public function of the package, as well as time spent executing the package without instrumentation, and using Jalangi2 without our provenance analysis (for benchmarking). "Triage rating" measures the average time spent executing the triage rating model on a generated provenance graph (Section 4.3). "Exploit synthesis" includes the time required to run the exploit and check whether it succeeded or failed (Section 4.2). Note that not every stage includes every package: 1) packages that timed out in a previous stage are not included in the subsequent one; 2) packages without a tainted flow also did not have triage rating or exploit synthesis performed.

D.3. Coordinated Vulnerability Disclosure

We follow a coordinated vulnerability disclosure process (i.e., responsible disclosure) [11] for the vulnerabilities discovered in our large-scale evaluation. This process is as follows: For potentially high-impact vulnerabilities, e.g., packages receiving around 100 downloads per week, we work with Snyk [87] to perform vulnerability disclosure. Snyk maintains its own disclosure timeline [86]; they contact package maintainers and work with us to explain and suggest remediation for vulnerabilities.

For packages with less than 100 downloads per week we directly contact package maintainers to explain the discovered vulnerabilities and suggest mitigations. Since many of these packages are unmaintained, we provide a 30 day response deadline with reminders sent to package maintainers. If we do not receive any communication from package maintainers within 30 days we report the vulnerability to CVE (MITRE) [20]. We allow package maintainers to request extensions of this timeline if they need additional time to patch the vulnerability.

D.4. Developer Response

In our coordinated vulnerability disclosure process (Appendix D.3), we directly or indirectly (through Snyk) contact developers to make them aware of discovered vulnerabilities. Developer responses have been minimal, even for packages with many downloads.

At the time of publication, we have received 7 responses from developers. All 7 developers have confirmed the reported vulnerability. Two developers chose to deprecate their package as a result of our reporting. Deprecating a npm package causes the package to be marked in the npm repository as "Deprecated", displaying a warning to users who attempt to install the package, but the package can still be installed and used [72]. Two developers chose to unpublish their package as a result of our reporting. Unpublishing a package delists it from the npm repository, making it impossible for new downloads of the package to occur [73]. Three developers attempted to produce a fix for the discovered vulnerability and asked for our review of the changes. These were cases of ACI where `exec` was used. In all cases, we guided the maintainers towards using `execFile` instead of `exec` as it does not spawn a shell by default [70].

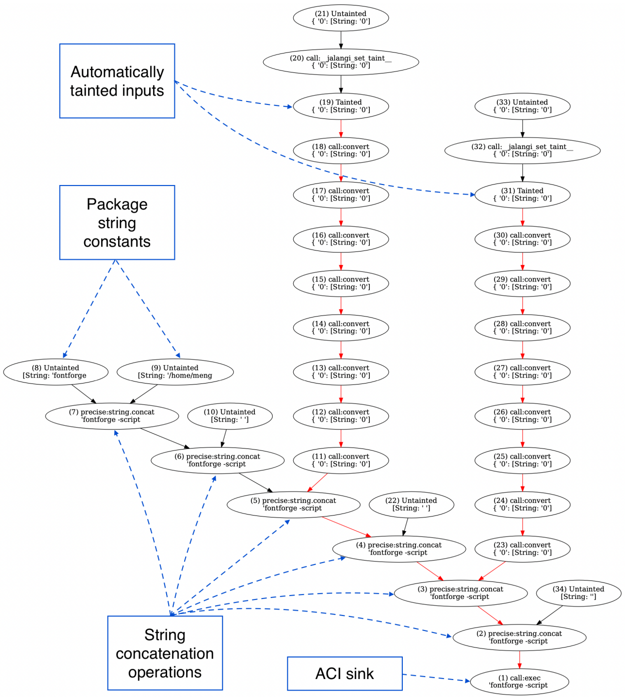

Automatically tainted inputs
Package string constants
String concatenation operations
ACI sink
(21) Untainted { '0': [String: '0']
(20) call:__jalangi_set_taint__ { '0': [String: '0']
(19) Tainted { '0': [String: '0']
(18) call:convert { '0': [String: '0']
(17) call:convert { '0': [String: '0']
(16) call:convert { '0': [String: '0']
(15) call:convert { '0': [String: '0']
(14) call:convert { '0': [String: '0']
(13) call:convert { '0': [String: '0']
(12) call:convert { '0': [String: '0']
(11) call:convert { '0': [String: '0']
(33) Untainted { '0': [String: '0']
(32) call:__jalangi_set_taint__ { '0': [String: '0']
(31) Tainted { '0': [String: '0']
(30) call:convert { '0': [String: '0']
(29) call:convert { '0': [String: '0']
(28) call:convert { '0': [String: '0']
(27) call:convert { '0': [String: '0']
(26) call:convert { '0': [String: '0']
(25) call:convert { '0': [String: '0']
(24) call:convert { '0': [String: '0']
(23) call:convert { '0': [String: '0']
(8) Untainted [String: 'fontforge
(9) Untainted [String: '/home/meng
(7) precise:string.concat 'fontforge -script
(10) Untainted [String: ' ']
(6) precise:string.concat 'fontforge -script
(5) precise:string.concat 'fontforge -script
(22) Untainted [String: ' ']
(4) precise:string.concat 'fontforge -script
(3) precise:string.concat 'fontforge -script
(34) Untainted [String: '']
(2) precise:string.concat 'fontforge -script
(1) call:exec 'fontforge -script

Figure 18. Provenance graph output for *font-converter*. Red arrows indicate tainted dataflow. Blue boxes and arrows are explanatory annotations. Attacker input flows (starting from nodes 21 and 33) along the red paths and is concatenated (at nodes 5, 4, 3, 2) with string constants from the package (nodes 8, 9, 22, 34) before reaching the sink (node 1).

D.5. Case Studies

Next, we present a selection of case studies of packages sourced from our large-scale evaluation.

font-converter. We presented this package previously in Section 2. Below we provide additional details to characterize this vulnerability, which is a straightforward example of arbitrary command injection (ACI) and has been recognized as a high-severity vulnerability with a CVSS score of 9.8, indicating critical severity[3].

The package provides a wrapper for the tool FontForge which allows for conversion between various font formats. The package consists of 13 lines of code with a total code size of 0.7 KB, and includes 1 dependency. NODEMEDIC's provenance analysis completed in 0.6 seconds and resulted in a provenance graph with 34 nodes and a depth of 16 nodes (Figure 18).

The provenance graph of this package showcases a typical pattern we see amongst ACI exploits; inputs to public APIs are passed unsanitized throughout the package, concatenated with package constant strings, and then directly given as a command to `exec`. Since `exec` spawns a shell [69], attacker-injected shell meta-characters in the command will be evaluated.

[3] https://security.snyk.io/vuln/SNYK-JS-FONTCONVERTER-2976194

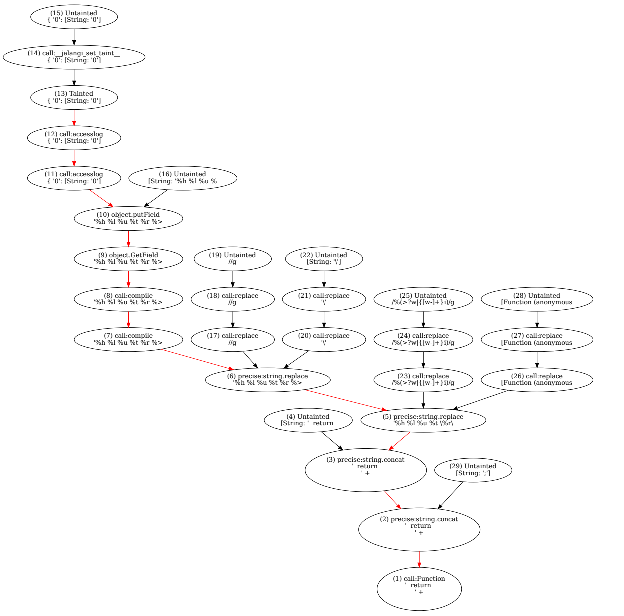


Figure 19. *accesslog* provenance graph.

Both the expert and model rating gave the package a high-exploitability rating. The package was able to be automatically confirmed by the exploit synthesis technique. The package was found to be vulnerable with arbitrary command injection because unsanitized user input was passed to the sink `exec`.

The automatically-generated exploit and driver were shown previously in Figure 3. The package's exported function accepts as its first two arguments a source and destination filepath (line 3). These arguments are passed directly to the call to `exec`, as shown previously in Figure 4, resulting in the aforementioned vulnerability. This vulnerability could be exploited if *font-converter* is used as a dependency of another package that accepts font files from users. If the user controls the name of the font file, that user will be able to inject arbitrary commands.

We reported this vulnerability to Snyk, who contacted the package maintainers but did not receive a response. Snyk disclosed this vulnerability as CVE-2022-21165 and gave it a CVSS score of 9.8, signaling critical severity.

accesslog. This package presents an example of an arbitrary command execution vulnerability that has been recognized as high-severity, with a CVSS score of 7.1[4]. Unlike the *font-converter* vulnerability, exploiting this vulnerability is not as straightforward, as shown below. Like many of the ACE vulnerabilities discovered by NodeMedic, this one could not be automatically confirmed because it requires careful setup of the package driver – in this case to simulate handling a HTTP request.

The package is intended to provide customizable logging middleware for HTTP request libraries. The package consists of 412 lines of code for a total code size of 13.6 KB, and includes 3 dependencies with a dependency tree depth of 1. NodeMedic's provenance analysis took 1.1 seconds to analyze the package. The provenance graph generated by the analysis is shown in Figure 19. It consists of 29 nodes with a depth of 14 nodes.

```
1   const accesslog = require("accesslog");
2   var handler = accesslog({
3       format: '\\\" + global.CTF();//',
4   });
5   var req = {};
6   var res = { end: function() {} };
7   handler(req, res, function() {});
8   res.end();
```

Figure 20. *accesslog* proof-of-concept driver.

```
1   var render = compile(options.format,
2     {options: options});
3   ...
4   function compile(format, context) {
5   ...
6   var js = '  return "' +
7   format.replace(/%(>?\w|{[\w-]+}i)/g,
8     function(_, name) {
9       return '"\n    +
10        (tokens["' + name + '"]
11          .call(this, req, res) || "-")
12        + "';
13    }) + '";';
14  return new Function(
15    'tokens, req, res', js).bind(context);
```

Figure 21. *accesslog* vulnerable code.

Both the expert and model ratings gave this package a rating of low-exploitability. Nonetheless, it was determined manually that this package is vulnerable to arbitrary code execution because the `Function` constructor is called with unsanitized user input.

This vulnerability occurs because the package accepts through its constructor a format string that is used in code generation (line 3 of Figure 20). When the resulting handler is generated by the package and then used to handle a (mock) request on line 7, the code generated from the format string is evaluated when the request's overwritten end function is called (line 8).

Inspecting the vulnerable code within the package (Figure 21), we see that the format string option value is passed, unsanitized, to the function `compile` (line 1), which interpolates it into a string (lines 6-13) used to construct a new function (lines 14-15). This vulnerability could be exploited if user input is used to inform the content of the format string, such as in the case of a server that exposes a web interface that allows for user-customizable formatting of monitored logs.

We reported this vulnerability through Snyk, who contacted the package maintainers but did not receive a response. Snyk disclosed this vulnerability as CVE-2022-25760 with a CVSS score of 7.1, signaling high severity.

comsvr-memory. In contrast to the *accesslog* case study, this package provides an example of an arbitrary code execution vulnerability that *could* be automatically confirmed by our exploit synthesis technique. Although both packages share a similar exploit string structure, save for the prefix of the string, what differentiates them is driver construction. *comsvr-memory* is able to be confirmed with the automatically-generated driver that directly calls the package's public APIs with the synthesized exploit.

The package provides a library of functions that cache data to local storage. The package consists of 507 lines of code for a total code size of 19.5 KB. The package has two dependencies and has a dependency tree depth of 1.

NodeMedic's analysis completed in 33.2 seconds for this package and produced a large provenance graph (Figure 22) with 142 nodes and a depth of 72 nodes. Both

[4] https://security.snyk.io/vuln/SNYK-JS-ACCESSLOG-2312099

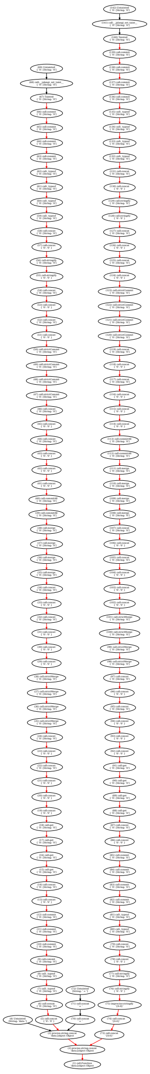


Figure 22. *comsvr-memory* provenance graph.

```
1   var PUT = require('comsvr-memory');
2   var x = "__proto__+global.CTF();//";
3   try {
4       var put = new PUT();
5       put.commit(x,x);
6   } catch (e) { console.log(e); }
```

Figure 23. Auto-generated *comsvr-memory* exploit driver.

the expert and model ratings for this package were low-exploitability, yet the exploit synthesized for the package successfully executed, confirming that an arbitrary command execution vulnerability was present.

The automatically generated exploit and driver code can be seen in Figure 23. On line 2, we see the exploit, which consists of the object prototype field name followed by binary addition and a call to our injected function `global.CTF`, whose execution indicates a successful exploit. On line 5, the package's `commit` function is called, which takes a name (key) and value to store.

```
1   commit(name, value) {
2   ...
3   } else if (typeof value === "string") {
4     new Function("data",
5     `return data.${name} = "${value}"`
6     )(this.data);
```

Figure 24. Vulnerable code within *comsvr-memory*.

The vulnerable code within *comsvr-memory* is show in Figure 24. On line 5 the package directly interpolates the `name` and `value` arguments into a string that is used to construct and evaluate a setter function on the data. Because the arguments are interpolated without sanitization, the exploit payload in the `name` argument gets interpreted as `return data.__proto__+global.CTF();//`. Since every object has the field `__proto__`, the field access succeeds. Subsequently, the attacker-controlled code (in this case the call to `global.CTF`) evaluates. This vulnerability could be exploited if another package uses *comsvr-memory* to cache user-submitted values.

When we went to report this vulnerability, we found the package had been unpublished after our large-scale evaluation completed; no further action was taken.

D.6. 10K Package Evaluation Tainted Flows

Tables 10 - 14 have anonymized[5] tainted flows from the 10K package evaluation. Columns descriptions follow.

Package characterization. "Name": Package's name in npm. "Version": Version at the time of gathering. Two metrics to measure maintenance and popularity: "Updated": Year the package last received an update. "LYDLs": Number of downloads in the past year (2022).

To quantify the complexity of the package: "Code Size (KB)": Size of JavaScript source files (including dependencies) in kilobytes. "LoC": Lines of code in JavaScript source files (including dependencies). "Num. Deps": Number of dependencies in dependency tree. "Dep. Depth": Depth of dependency tree.

Analysis output. We included metrics to quantify analysis runtime and output: "Analysis Time": Total analysis runtime in seconds. "Rating": Triage rating model ouput (Section 4.3). "Auto-conf.": Whether automatic confirmation succeeded (Section 4.2).

We also quantify the flow complexity with provenance graph metrics: "Prov. Nodes": Number of nodes in the provenance graph. "Prov. Depth": Depth of the provenance graph (length of longest path).

Vulnerability characterization. "Vuln.": Type of vulnerability found, if any. "Vuln. Deps.": Number of dependencies with code executed in the vulnerability trace. "Vuln. Depth": Depth of vulnerable sink in dependency tree (package itself is depth 0).

The above metrics, in addition to the previously described "Prov. Nodes" and 'Prov. Depth', quantify vulnerability complexity. "Vuln. Depth" does not necessarily indicate complexity, but shows where sinks are located in the package's dependency tree.

Finally, we report on the disclosure process at the time of this paper's publication. "Status": The disclosure status with values: R: Report filed to Snyk or the package maintainer. D: Vulnerability disclosed. U: Package unpublished [73] from npm. "CVE": The CVE assigned to the vulnerability, if any. CVEs marked with † were not disclosed by our work.

[5] Responsible disclosure (Appendix D.3) is ongoing at the time of publication; only disclosed vulnerabilities are deanonymized.

TABLE 10. 10K PACKAGE TAINTED FLOW RESULTS (1/5)

Name	Version	Updated	LY-DLs	Code Size	LoC	Num. Deps.	Dep. Depth	Analysis Time	Prov. Nodes	Prov. Depth	Rating	Auto-conf.	Vuln.	Vuln. Deps.	Vuln. Depth	Status	CVE
a*****	0.0.2	2012	1361	7.5	360	2	1	42.2	101	28	L	-	ACE	0	0	R	-
accesslog	0.0.2	2013	3865	13.6	412	3	1	1.1	29	14	L	-	ACE	2	0	D	CVE-2022-25760
a*****	0.0.1	2013	133	24.3	1097	3	2	12.4	35	33	H	✓	ACI	2	0	R	-
a*****	1.0.3	2020	258	0.4	13	1	0	0.5	6	6	H	✓	ACI	0	0	R	-
a*****	1.0.3	2016	989	2.2	99	1	0	0.9	33	21	H	-	ACI	0	0	R	-
a*****	1.4.9	2019	133	2.6	75	1	0	1.0	73	71	H	✓	ACI	0	0	R	-
a*****	1.0.0	2020	110	1.9	1	1	0	1.0	78	24	M	✓	ACI	0	0	R	-
a*****	0.0.3	2021	199	1443.5	45116	3	1	2.9	12	12	H	-	ACI	2	0	R	-
a*****	0.1.14	2020	4337	313.3	367	14	4	6.1	137	61	L	✓	ACI	0	0	R	-
a*****	1.0.1	2021	140	8.7	240	1	0	3.6	24	24	L	-	ACE	0	0	R	-
a*****	0.0.8	2015	73318	2.7	73	1	0	0.8	31	28	H	✓	ACI	0	0	R	-
a*****	1.0.4	2016	303	90.4	3179	2	1	1.0	12	9	H	✓	ACI	0	0	R	-
a*****	1.0.8	2015	568	1.3	34	1	0	0.8	12	9	H	✓	ACI	0	0	R	-
a*****	1.0.7	2017	283	4.4	136	1	0	0.8	12	9	H	✓	ACI	0	0	R	-
a*****	1.0.2	2017	257	216.4	7418	5	2	9.7	49	39	H	✓	ACI	2	0	R	-
a*****	0.1.0	2016	113	3.3	95	1	0	0.8	8	7	H	✓	ACI	0	0	R	-
a*****	1.1.0	2017	33509	2.8	52	1	0	1.7	41	37	H	✓	ACI	0	0	R	-
atat	1.2.12	2021	740	49.1	1095	1	0	123.6	6724	77	M	-	-	-	-	-	-
a*****	1.0.4	2018	270	12.4	65	1	0	0.8	33	16	H	-	ACI	0	0	R	-
a*****	1.0.2	2016	260	1.5	50	2	1	0.5	20	10	H	✓	ACI	1	0	R	-
a*****	0.1.3-2	2013	419	7.3	222	1	0	0.7	11	11	H	-	ACI	0	0	R	-
b*****	1.0.0	2021	96	315.1	8280	21	6	239.8	26	26	H	✓	ACI	12	1	R	-
b*****	1.0.0	2013	111	35.9	1178	2	1	0.8	34	18	H	✓	ACI	1	0	R	-
b*****	0.0.3	2014	175	0.6	18	1	0	0.5	26	14	H	-	ACI	0	0	R	-
b*****	0.1.0	2018	159	7.5	216	1	0	1.0	36	13	M	-	ACE	0	0	R	-
b*****	1.0.0	2019	99	1.2	22	1	0	0.6	20	11	M	✓	ACI	0	0	R	-
blue-data-model	1.0.2	2018	0	21.6	619	1	0	44.7	12	10	H	-	ACE	0	0	U	-
brainfuck-compiler	1.0.6	2016	459	10.5	393	1	0	1.5	208	33	H	-	-	-	-	-	-
broccoli-compass	0.2.4	2015	669	505.7	18986	19	5	158.1	29	19	H	✓	ACI	17	0	D	CVE-2023-27848
c*****	0.1.0	2012	390	190.9	4343	5	2	0.7	11	11	H	-	ACI	1	0	R	-
c*****	1.1.0	2016	259	1.2	44	1	0	0.6	6	6	H	✓	ACI	0	0	R	-
c*****	1.5.2	2015	647	16.1	765	2	1	4.1	51	30	M	-	ACE	1	0	R	-
c*****	0.0.7	2014	386	14.4	545	3	2	2.3	8	8	H	✓	ACI	2	0	R	-
comsvr-memory	0.1.0	2022	0	19.5	507	2	1	33.2	142	72	L	✓	ACE	1	0	U	-
d*****	0.3.10	2016	601	17.3	544	1	0	3.8	30	29	M	✓	ACI	0	0	R	-
d*****	1.3.0	2014	354	1003.3	38157	37	5	76.7	68	34	L	-	ACE	2	0	R	-
enpeem	2.2.0	2016	55872	17.8	360	3	2	1.6	705	22	H	-	ACI	2	0	D	CVE-2019-10801†
e*****	1.0.4	2016	239	5.1	119	1	0	1.8	20	20	H	✓	ACI	0	0	R	-

TABLE 11. 10K PACKAGE TAINTED FLOW RESULTS (2/5)

Name	Version	Updated	LY-DLs	Code Size	LoC	Num. Deps.	Dep. Depth	Analysis Time	Prov. Nodes	Prov. Depth	Rating	Auto-conf.	Vuln.	Vuln. Deps.	Vuln. Depth	Status	CVE
error-foundry-js	0.1.0	2014	121	2.5	54	1	0	0.7	58	20	H	-	-	-	-	-	-
e*****	0.0.2	2016	168	791.7	21752	19	7	28.6	9	9	H	✓	ACI	11	0	R	-
e*****	1.0.2	2016	192	6.1	145	1	0	1.2	30	30	L	✓	ACE	0	0	R	-
e*****	2.2.0	2015	335	74.5	2420	2	1	2.4	42	40	H	-	ACI	1	0	R	-
e*****	1.0.2	2023	424	1.9	71	1	0	0.7	14	14	M	-	ACI	0	0	R	-
e*****	1.0.1	2016	20018	1.1	30	1	0	0.5	6	6	H	✓	ACI	0	0	R	-
e*****	1.0.0	2016	4759	0.6	18	1	0	0.5	8	8	H	-	ACI	0	0	R	-
e*****	1.1.2	2022	378	2.0	28	1	0	0.8	32	32	M	✓	ACI	0	0	R	-
extra-function	0.0.51	2023	23862	11.0	237	1	0	3.5	4	4	H	-	-	-	-	-	-
e*****	19.6.0	2018	810	3.8	91	1	0	2.4	210	39	L	-	ACI	0	0	R	-
e*****	0.0.31	2016	363	2189.4	61428	58	6	2.6	16974	91	M	✓	ACI	0	0	R	-
f*****	1.0.1	2015	224	0.5	5	1	0	0.5	24	13	H	✓	ACI	0	0	R	-
font-converter	1.1.1	2015	1779	0.7	13	1	0	0.6	34	16	H	✓	ACI	0	0	D	CVE-2022-21165
fool-node	1.0.0	2021	124	1.1	48	1	0	0.9	4	4	H	-	-	-	-	-	-
f*****	0.0.2	2013	147	1.8	27	1	0	0.5	12	10	H	-	ACI	0	0	R	-
f*****	1.0.0	2020	103	1.0	21	1	0	0.7	10	9	M	✓	ACI	0	0	R	-
f*****	1.0.8	2021	398	11.6	258	2	1	12.6	94	85	H	-	ACI	0	0	R	-
f*****	0.0.2	2012	139	4.3	59	1	0	0.7	56	29	M	✓	ACI	0	0	R	-
future-proxy	1.2.0	2021	212	29.7	487	2	1	8.1	40	34	M	-	-	-	-	-	-
g*****	0.0.7	2014	510	62.9	1439	2	1	0.8	8	8	H	✓	ACI	0	0	R	-
g*****	1.0.0	2015	227	6.3	183	2	1	1.0	48	22	H	✓	ACI	0	0	R	-
gep	3.0.0	2016	424	17.8	464	1	0	6.7	18	17	M	-	-	-	-	-	-
g*****	0.0.2	2015	116	46.8	928	2	1	2.0	17	17	H	✓	ACI	0	0	R	-
g*****	1.0.3	2018	120234	475.4	11012	55	10	8.3	408	54	M	-	ACI	8	2	R	-
g*****	0.0.1	2018	130	340.7	9586	52	10	6.3	408	54	M	-	ACI	8	2	R	-
h*****	0.2.1	2016	33187	75.8	1256	4	3	5.5	43	43	M	-	ACE	3	0	R	-
h*****	0.0.1	2014	117	1.4	52	1	0	0.7	4	4	H	✓	ACI	0	0	R	-
h*****	0.0.2	2015	125	41.1	1307	2	1	5.4	27	27	H	✓	ACI	1	0	R	-
h*****	1.5.12	2019	1618	12.8	302	1	0	2.3	37	37	L	✓	ACI	0	0	R	-
h*****	1.0.2	2017	159	60.5	2857	9	2	316.1	128	38	L	-	ACE	8	0	R	-
hoopoe	1.0.3	2021	317	3.1	73	1	0	0.7	19	18	H	✓	ACI	0	0	U	-
h*****	0.0.1	2018	104	23.2	923	4	2	2.0	6	6	H	✓	ACI	3	0	R	-
hot	0.0.7	2013	9764	26.5	288	1	0	1.8	340	46	L	-	-	-	-	-	-
huedawn-tesseract	0.3.3	2019	177	99.7	3699	12	4	36.4	749	26	H	✓	ACI	11	0	U	-
h*****	1.0.10	2019	403	3.3	57	1	0	0.8	16	16	H	✓	ACI	0	0	R	-
i*****	1.0.2	2018	150	17.5	598	1	0	22.1	14	14	H	✓	ACI	0	0	R	-
i*****	1.0.2	2017	5066	18.4	466	2	1	0.8	24	16	H	✓	ACI	1	0	R	-
l*****	1.0.4	2017	246	49.2	908	9	4	1.6	1540	1538	M	✓	ACI	4	0	R	-
l*****	0.0.5	2014	513	3.7	122	1	0	0.9	24	22	H	✓	ACI	0	0	R	-
l*****	1.0.2	2019	159	3.8	28	1	0	0.6	8	7	H	✓	ACI	0	0	R	-

TABLE 12. 10K PACKAGE TAINTED FLOW RESULTS (3/5)

Name	Version	Updated	LY-DLs	Code Size	LoC	Num. Deps.	Dep. Depth	Analysis Time	Prov. Nodes	Prov. Depth	Rating	Auto-conf.	Vuln.	Vuln. Deps.	Vuln. Depth	Status	CVE
l*****	1.0.0	2018	352	0.8	39	1	0	0.6	17	16	M	✓	ACI	0	0	R	-
l*****	1.0.6	2016	267	32.4	1360	2	1	0.8	20	20	M	✓	ACI	0	0	R	-
l*****	1.0.0	2019	117	1.2	32	1	0	0.9	16	16	H	✓	ACI	0	0	R	-
l*****	0.0.4	2015	165	143.5	5207	15	4	7.7	101	27	H	-	ACI	6	0	R	-
l*****	0.0.1	2017	208	27.2	376	1	0	1.5	168	26	H	-	ACI	0	0	R	-
list-git-branches	1.0.0	2017	3021	2.0	94	3	2	1.0	149	42	H	-	-	-	-	-	-
list-git-remotes	1.0.1	2017	42760	2.2	111	3	2	0.8	2	2	H	-	-	-	-	-	-
l*****	0.0.16	2020	456	132.7	4636	14	5	199.1	316	216	L	✓	ACI	13	0	R	-
l*****	1.0.0	2016	102	10.5	234	3	1	1.0	14	14	H	✓	ACI	2	0	R	-
loda	0.1.3	2014	399	64.8	2070	1	0	141.2	933	105	L	-	-	-	-	-	-
l*****	1.5.0	2019	146	1.8	35	1	0	0.6	20	20	H	✓	ACI	0	0	R	-
l*****	0.1.0	2017	160	18.1	600	2	1	1.0	11	11	H	✓	ACI	0	0	R	-
l*****	1.1.2	2018	223	1.0	18	1	0	0.6	12	11	H	✓	ACE	0	0	R	-
l*****	0.0.4	2015	193	613.2	16233	3	2	1.2	26	14	H	✓	ACI	0	0	R	-
lzc-node	1.0.7	2019	259	109.2	3502	8	2	45.8	144	144	L	-	-	-	-	-	-
m*****	0.1.0	2019	98	87.7	2444	8	3	2.6	26	25	M	✓	ACI	2	0	R	-
macfromip	1.1.1	2015	6639	6.7	170	1	0	1.2	30	28	H	✓	ACI	0	0	D	CVE-2020-7786†
macos	0.0.1	2016	1290	5.5	106	1	0	0.8	55	17	L	-	-	-	-	-	-
m*****	0.1.0	2016	133	2.7	37	1	0	0.7	26	13	L	✓	ACE	0	0	R	-
m*****	0.1.0	2017	173	0.9	30	1	0	0.5	22	10	H	✓	ACI	0	0	R	-
m*****	1.0.1	2017	49145	6.0	165	1	0	1.1	20	18	M	-	ACE	0	0	R	-
m*****	0.0.4	2018	178	6.7	146	1	0	2.6	50	37	M	✓	ACI	0	0	R	-
m*****	0.0.7	2014	398	605.3	16983	2	1	10.6	874	173	H	✓	ACI	0	0	R	-
m*****	1.0.7	2021	274	1405.2	32039	131	6	0.5	12	7	H	✓	ACI	73	0	R	-
m*****	0.1.7	2017	1443	48.8	1225	1	0	9.2	61	59	M	-	ACE	0	0	R	-
m*****	2.0.5	2016	260	5.0	144	1	0	1.1	5	5	H	-	ACI	0	0	R	-
moform-utils	1.1.7	2022	0	9.1	1	1	0	3.5	60	59	M	✓	ACE	0	0	U	-
m*****	0.1.1	2015	335	1.2	47	1	0	0.6	20	19	H	✓	ACI	0	0	R	-
n*****	1.0.1	2016	151	2.7	69	1	0	1.1	28	23	H	✓	ACI	0	0	R	-
n*****	0.0.1	2013	116	0.9	28	1	0	0.6	50	18	H	✓	ACI	0	0	R	-
node-atpl	1.1.5	2017	255	52.0	1812	1	0	7.6	52	34	M	-	-	-	-	-	-
n*****	0.0.5	2017	195	2.6	81	1	0	0.6	9	8	H	-	ACI	0	0	R	-
n*****	1.0.0	2017	112	0.9	22	1	0	0.6	14	13	M	✓	ACI	0	0	R	-
n*****	0.2.0	2020	128	1458.2	45495	3	1	13.3	2	2	H	-	ACI	0	0	R	-
n*****	1.0.7	2018	339	2.5	44	1	0	1.4	20	15	M	-	ACI	0	0	R	-
n*****	0.1.3	2015	187	1.0	17	1	0	0.9	61	23	H	✓	ACI	0	0	R	-
n*****	0.0.3	2014	254	547.9	14736	2	1	1.1	22	13	H	✓	ACI	0	0	R	-
n*****	0.0.1	2015	118	1.3	34	1	0	0.6	27	13	H	✓	ACI	0	0	R	-
n*****	0.0.5	2019	358	1.5	50	1	0	0.7	14	11	H	✓	ACI	0	0	R	-
p*****	1.0.0	2020	444	3.0	86	1	0	0.8	58	30	M	✓	ACI	0	0	R	-

TABLE 13. 10K PACKAGE TAINTED FLOW RESULTS (4/5)

Name	Version	Updated	LY-DLs	Code Size	LoC	Num. Deps.	Dep. Depth	Analysis Time	Prov. Nodes	Prov. Depth	Rating	Auto-conf.	Vuln.	Vuln. Deps.	Vuln. Depth	Status	CVE
rails-routes-to-json	1.0.0	2017	124	3.1	95	1	0	0.9	14	14	H	✓	ACI	0	0	D	CVE-2023-27849
r*****	1.1.0	2018	160	4.6	154	1	0	2.7	14	14	H	✓	ACI	0	0	R	-
r*****	0.1.4	2014	405	23.9	373	1	0	2.6	11100	65	L	-	ACE	0	0	R	-
r*****	1.0.3	2019	13898	1.0	26	1	0	0.6	55	27	M	✓	ACI	0	0	R	-
r*****	0.4.0	2016	233	6.8	132	1	0	1.3	190	26	L	-	ACE	0	0	R	-
r*****	1.0.1	2018	151	3748.6	90388	3	1	1.7	106	16	M	-	ACI	0	0	R	-
r*****	1.0.2	2017	181	4.0	179	1	0	5.4	117	34	H	-	ACI	0	0	R	-
redux-app	2.1.0	2018	1137	2290.4	72391	9	3	72.1	37246	176	L	-	-	-	-	-	-
r*****	0.4.7	2014	249	589.1	16137	2	1	17.4	36	19	H	✓	ACI	0	0	R	-
r*****	0.0.4	2017	201	2.6	83	1	0	0.5	65	27	L	-	ACE	0	0	R	-
r*****	1.0.1	2017	159	2.1	64	1	0	0.8	6	6	H	✓	ACI	0	0	R	-
r*****	1.0.0	2018	152	0.4	7	1	0	0.5	6	6	M	✓	ACE	0	0	R	-
s*****	0.1.2	2018	172	296.7	10862	62	8	1.2	2	2	H	-	ACI	1	0	R	-
s*****	0.1.1	2014	10348	12.3	193	1	0	16.7	40	40	H	✓	ACI	0	0	R	-
s*****	1.0.2	2016	173	1468.2	45864	3	1	1.8	71	53	H	✓	ACI	2	1	R	-
s*****	1.0.1	2018	141	4.5	154	1	0	3.1	12	12	H	✓	ACI	0	0	R	-
s*****	1.0.2	2018	181	17.5	382	1	0	24.6	2	2	H	-	ACI	0	0	R	-
s*****	1.0.1	2018	122	0.5	16	1	0	0.5	8	8	H	✓	ACI	0	0	R	-
s*****	0.0.4	2014	207	68.1	1421	3	1	1.0	625	79	M	✓	ACI	0	0	R	-
s*****	0.0.6	2014	232	109.9	3885	3	1	35.2	26	24	H	✓	ACI	1	0	R	-
s*****	1.0.21	2021	684	2.9	90	1	0	0.7	72	21	H	-	ACI	0	0	R	-
s*****	0.0.1	2014	86731	1.0	31	1	0	0.5	26	14	H	✓	ACI	0	0	R	-
s*****	2.0.0	2014	167	11.4	234	1	0	1.7	186	32	H	-	ACI	0	0	R	-
s*****	0.0.1	2017	123	0.7	10	1	0	0.7	8	7	M	✓	ACI	0	0	R	-
s*****	0.1.0	2014	1482	1.3	55	1	0	0.6	10	9	H	✓	ACI	0	0	R	-
s*****	0.0.2	2019	97	2346.0	73114	3	1	3.4	88	44	H	-	ACI	1	0	R	-
s*****	0.1.4	2014	200	681.3	18981	7	2	21.7	46	32	H	✓	ACI	5	0	R	-
s*****	1.1.8	2019	640	103.5	3843	7	2	49.6	96	96	M	✓	ACI	4	0	R	-
s*****	0.0.2	2011	681	22.5	1004	2	1	3.4	34	18	H	✓	ACI	0	0	R	-
s*****	1.3.0	2018	297	1.1	40	1	0	0.5	53	16	H	-	ACI	0	0	R	-
smartctl	1.0.0	2015	124	4.1	123	1	0	0.9	26	24	H	✓	ACI	0	0	D	CVE-2022-21810†
s*****	0.0.1	2018	133	1.7	43	1	0	0.8	9	9	H	✓	ACI	0	0	R	-
s*****	1.4.0	2016	900	1.1	53	1	0	0.6	106	20	H	✓	ACI	0	0	R	-
s*****	1.0.2	2018	242	6.8	86	1	0	1.0	278	62	H	✓	ACI	0	0	R	-
s*****	0.1.6	2020	348	49392.9	37	51	7	0.8	30	16	M	✓	ACI	0	0	R	-
s*****	1.3.3	2021	5437	3.8	105	1	0	8.0	16	16	H	✓	ACI	0	0	R	-
s*****	1.0.2	2020	159	1.7	43	2	1	1.0	9	9	H	✓	ACI	0	0	R	-
s*****	0.0.1	2016	94	1.1	16	1	0	0.7	14	13	H	✓	ACI	0	0	R	-
t*****	0.1.6	2011	2107	149.8	3305	2	1	25.6	2079	210	L	✓	ACI	1	0	R	-

TABLE 14. 10K PACKAGE TAINTED FLOW RESULTS (5/5)

Name	Version	Updated	LY-DLs	Code Size	LoC	Num. Deps.	Dep. Depth	Analysis Time	Prov. Nodes	Prov. Depth	Rating	Auto-conf.	Vuln.	Vuln. Deps.	Vuln. Depth	Status	CVE
t*****	0.1.0	2014	122	621.1	16683	7	3	5.9	10	10	H	✓	ACI	4	0	R	-
t*****	0.1.0	2014	127	1035.4	28569	4	2	2.7	142	39	H	-	ACI	2	0	R	-
t*****	1.0.1	2018	126	1.2	47	1	0	0.6	12	12	H	✓	ACI	0	0	R	-
t*****	0.0.8	2019	307	2.2	94	1	0	0.7	6	6	H	-	ACI	0	0	R	-
t*****	1.0.0	2013	289	1.1	38	1	0	0.6	33	16	H	✓	ACI	0	0	R	-
text-privacy-converter	1.0.0	2018	123	1.1	18	1	0	2.3	404	20	H	-	-	-	-	-	-
the-first-commit	0.0.1	2018	113	2.6	123	2	1	0.5	2	2	H	-	-	-	-	-	-
t*****	0.0.2	2018	178	3.0	106	1	0	0.7	12	9	M	✓	ACI	0	0	R	-
t*****	0.1.1	2016	123	2618.5	91608	145	14	1.6	20	19	H	✓	ACI	2	0	R	-
t*****	1.3.0	2022	2707372	276.2	7057	27	6	1.6	34	34	L	✓	ACI	0	0	R	-
t*****	1.0.0	2016	106	36.2	1178	2	1	0.9	34	18	H	✓	ACI	1	0	R	-
t*****	2.0.2	2020	26225	28.2	231	1	0	9.0	24	24	H	✓	ACI	0	0	R	-
t*****	0.1.0	2013	172	2.5	54	1	0	1.3	19	18	H	✓	ACI	0	0	R	-
t*****	1.0.1	2021	146	41.5	179	2	1	0.8	26	12	L	-	ACE	0	0	R	-
to-clipboard-android	0.2.0	2016	169	0.9	49	1	0	0.6	2	2	H	-	-	-	-	-	-
t*****	1.1.2	2020	1003	14.0	296	1	0	0.7	35	25	L	-	ACE	0	0	R	-