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

Darion Cassel, Wai Tuck Wong, Limin Jia
Introduction: Node.js JavaScript Runtime

Node.js is widely used for server-side, desktop, and IoT development

npm: Ecosystem of 1 million+ packages developers can use
Node.js is Popular for Attackers Too

Node.js package vulnerabilities in the news

NPM package with 3 million weekly downloads had a severe vulnerability

Someone slipped a vuln into crypto-wallets via an NPM package. Then someone else siphoned off $13m in coins

Have you updated your Electron app? We hope so. There was a bad code-injection bug in it

GitHub security team finds remote code execution bug in popular Node.js changelog library
**Background: Node.js Package Attacker Model**

**Attack:**
1) Submits exploit to PA  
2) PA passes exploit to Dep 6  
3) Dep 6 passes exploit to `exec`

**PA** includes a vulnerable dependency (Dep 6)

PA has an unsanitized dataflow (heavy red arrows) to Dep 6

Dep 6 accesses privileged Node.js API (`exec`)

**Arbitrary Code Execution (ACE; CWE-94)**

**Arbitrary Command Injection (ACI; CWE-88)**

From npm
Background: Node.js Package Attacker Model

Prior work detects these flows with dynamic taint analysis

![Diagram showing the flow of dependencies with an exploit]

Challenge: Average package has 79 dependencies to be checked [Zimmerman 2019]


Challenges for Node.js Package Dynamic Taint Analysis

1. Driving package APIs
2. Precise analysis of built-in datatypes
3. Scaling to large dependency trees
4. Triage of tainted flows
5. Confirmation of tainted flows

End-to-End Analysis Infrastructure

Provenance Graphs
Augmenting Taint Analysis with Provenance Graphs

Provenance Tracking

- Prior: Policy-based taint propagation
- **Graph** of operations performed

Foundation for further analysis:

- Exploit synthesis (*covered later*)
- Triage (*see paper*)

NodeMedic: End-to-End Analysis of Node.js Vulnerabilities with Provenance Graphs
NodeMedic End-to-End Analysis Infrastructure (1/2)

Challenges: 1) Driving package APIs  2) Precise analysis  3) Scalable analysis
NodeMedic End-to-End Analysis Infrastructure (2/2)

Challenges: 4, 5) Reduce analyst triage burden
Solution: Scalable Analysis of Large Dependency Trees

Motivation: Packages avg 79 deps

Insight: Not every dependency needs precise analysis; deeper deps. don’t add flows but increase overhead

Algorithm: Mark, based on a package’s depth in tree, whether to analyze precisely or imprecisely

Tuning: Analyst-controllable parameters w.r.t. tree size & depth
**Solution: Reducing Analyst Triage Burden (1/2)**

**Motivation:** Analyst must manually *confirm* reported tainted flows
- *Confirm:* Construct a proof-of-concept (PoC) exploit
- Reduces analysis scalability

**Insight:** Provenance graph contains operations performed on tainted value

```javascript
// Driver Code
__set_taint__('tainted');
grep('tainted');
```

```javascript
// Package API
function grep(inpt) {
  exec('grep ' + inpt);
}
```

1. call:*exec* `'grep tainted'`
2. Untainted
   - [String: 'grep ']
3. Untainted
   - [String: 'tainted']
4. Tainted
   - [String: 'tainted']
5. call:*grep* 'tainted'
   - Untainted
6. Untainted
   - [String: 'tainted']
7. call:*__set_taint__* 'tainted'
8. Untainted
   - [String: 'tainted']
Solution: Reducing Analyst Triage Burden (2/2)

① Provenance graph → SMT formula encoding operations and PoC

② Solve with Z3 and derive model if SAT

③ Rerun package with candidate PoC

④ Check for PoC success
Results: Large-Scale Evaluation on Real Node.js Packages

**Result:** Scalable analysis of 10,000 packages from npm

**Prior work:** ~20 packages [1, 2]

<table>
<thead>
<tr>
<th>Package Results</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inherent package issues</td>
<td>394</td>
</tr>
<tr>
<td>Package analysis timeout</td>
<td>258</td>
</tr>
<tr>
<td>No tainted flows</td>
<td>9175</td>
</tr>
<tr>
<td>Tainted flows</td>
<td>173</td>
</tr>
</tbody>
</table>

**Result:** Able to automatically confirm 108 potential flows

<table>
<thead>
<tr>
<th>Type</th>
<th>Count</th>
<th>Confirmed</th>
<th>Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arbitrary command injection (ACI)</td>
<td>133</td>
<td>102</td>
<td>76%</td>
</tr>
<tr>
<td>Arbitrary code execution (ACE)</td>
<td>22</td>
<td>6</td>
<td>27%</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>155</td>
<td>108</td>
<td>70%</td>
</tr>
</tbody>
</table>
More in the Paper and our Repository

In the paper:

- Precise provenance analysis
- Custom propagation policies
- Triage rating methodology

→ github.com/NodeMedicAnalysis

- End-to-end infrastructure
- 589 taint precision tests
- Case studies