SecBench.js

An Executable Security Benchmark Suite for Server-Side JavaScript

Michael Pradel, University of Stuttgart
Joint work with Masudul Hasan Masud Bhuiyan, Adithya Srinivas Parthasarathy, Nikos Vasilakis, and Cristian-Alexandru Staicu
Why Do We Want Benchmarks?

- **Fuels progress in a research community**
  - E.g., MNIST in machine learning, SPEC CPU in compilers

- **Avoids duplicate work**
  - Gathering and setting up a dataset takes time

- **Makes approaches comparable**
  - Head-to-head comparison, instead of “we believe we are better because…”
Focus: JavaScript Vulnerabilities

- **Scope**
  - JavaScript packages on npm
  - Server-side code
  - Vulnerable (not malicious) code

- **Importance**
  - > 2 million npm packages
  - Thousands of vulnerabilities
  - Dozens of new vulnerability-related techniques each year
Example: Command Injection

Vulnerable code (bestzip package):

```javascript
const command = 'zip --quiet --recurse-paths ${options.destination} ${sources}';
const zipProcess = cp.exec(command, {
  stdio: "inherit",
  cwd: options.cwd
});
```

Untrusted string becomes part of an OS-level command
Example: Command Injection

Vulnerable code (bestzip package):

```javascript
const command = 'zip --quiet --recurse-paths ${
  options.destination
} ${sources}';
const zipProcess = cp.exec(command, {
  stdio: "inherit",
  cwd: options.cwd
});
```

Untrusted string becomes part of an OS-level command

Attack code:

```javascript
zip({
  source: "",
  destination: ".;/ touch bestzip",
});
```

Attacker can execute arbitrary commands
Desired Properties of a Benchmark

- Realistic
- Executable
- Two-sided
- Vetted
Desired Properties of a Benchmark

- Realistic
- Executable
- Two-sided
- Vetted

- Diverse, real-world software
- Unmodified code
- Why?
  - Success on benchmark
  ⇒ Success on reality
 Desired Properties of a Benchmark

- Realistic
- Executable
- Two-sided
- Vetted

- Proof-of-concept attack that exploits the vulnerability
- Why?
  - Evidence that exploitable
  - Basis for evaluating mitigation techniques
Desired Properties of a Benchmark

- Realistic
- Executable
- Two-sided
- Vetted

- Both vulnerable and fixed code
- Why?
  - Evaluate false positives
  - Study and learn from fixes
Desired Properties of a Benchmark

- Realistic
- Executable
- Two-sided
- Vetted

Why?
- Avoid noise of large-scale, automated data gathering
## Existing Benchmarks

<table>
<thead>
<tr>
<th>Benchmark/dataset</th>
<th>Language</th>
<th>Vulns.</th>
<th>Realistic Exec. exploits</th>
<th>Two-sided</th>
<th>Vetted</th>
</tr>
</thead>
<tbody>
<tr>
<td>CGC</td>
<td>C</td>
<td>590</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Juliet</td>
<td>C/C++, Java, C#</td>
<td>121,922</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>LAVA-M</td>
<td>C</td>
<td>2,265</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>BigVul</td>
<td>C/C++</td>
<td>3,745</td>
<td>✓</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Ferenc et al. ’19</td>
<td>JavaScript</td>
<td>1,496</td>
<td>✓</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>VulinOSS</td>
<td>various</td>
<td>17,738</td>
<td>✓</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Magma</td>
<td>C</td>
<td>118</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>Ghera</td>
<td>Java/Android</td>
<td>25</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>Ponta et al.</td>
<td>Java</td>
<td>624</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
</tr>
</tbody>
</table>
## Existing Benchmarks

<table>
<thead>
<tr>
<th>Benchmark/dataset</th>
<th>Language</th>
<th>Vulns.</th>
<th>Realistic</th>
<th>Exec. exploits</th>
<th>Two-sided</th>
<th>Vetted</th>
</tr>
</thead>
<tbody>
<tr>
<td>CGC</td>
<td>C</td>
<td>590</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Juliet</td>
<td>C/C++, Java, C#</td>
<td>121,922</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>LAVA-M</td>
<td>C</td>
<td>2,265</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>BigVul</td>
<td>C/C++</td>
<td>3,745</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Ferenc et al. ’19</td>
<td>JavaScript</td>
<td>1,496</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>VulinOSS</td>
<td>various</td>
<td>17,738</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Magma</td>
<td>C</td>
<td>118</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Ghera</td>
<td>Java/Android</td>
<td>25</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Ponta et al.</td>
<td>Java</td>
<td>624</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SecBench.js</td>
<td>JavaScript</td>
<td>600</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
SecBench.js

■ 600 JavaScript vulnerabilities

☐ Code injection
☐ Command injection
☐ Path traversal
☐ Prototype pollution
☐ ReDoS

■ Three applications

See ICSE’23 paper and https://github.com/cristianstaicu/SecBench.js
Methodology

Three data sources:
Snyk, GitHub Advisories, Huntr.dev

Filter: Available, installable, reproducible

Create exploits

Search for CVE and fixing commit
Creating Exploits

- Validate that code is **vulnerable and can be exploited**

- **Two steps:**
  1) Perform **security-relevant action**
  2) Check success with **exploit oracle**
Creating Exploits

- Validate that code is **vulnerable and can be exploited**

- **Two steps:**
  1) Perform security-relevant action
  2) Check success with exploit oracle

**Example: Code and command injection**

1) Create file
2) Check whether file exists
Creating Exploits

- Validate that code is **vulnerable and can be exploited**

- **Two steps:**
  1) Perform security-relevant action
  2) Check success with exploit oracle

Example: ReDoS

1) Trigger expensive regexp matching
2) Check that processing time > threshold
Creating Exploits

- Validate that code is **vulnerable** and **can be exploited**

- Two steps:
  1) Perform **security-relevant action**
  2) Check success with **exploit oracle**

**Example: Prototype polution**

1) Add special property to prototype of all objects
2) Check that property exists
Example: Prototype Pollution

test("prototype pollution in lodash", () => {
  // setup
  const mergeF = require("lodash").defaultsDeep;
  const payload = '{"constructor": {"prototype": {"polluted": "yes"}}}';
  // sanity check
  expect({}.polluted).toBe(undefined);
  // exploit
  mergeF({}, JSON.parse(payload));
  // oracle check
  expect({}.polluted).toBe("yes");
  // cleanup
  delete Object.prototype.polluted;
});
## Overview of Benchmark

<table>
<thead>
<tr>
<th>Type of vulnerability</th>
<th>Nb. exploits</th>
<th>Has fix</th>
<th>Has CVE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code injection</td>
<td>40</td>
<td>21</td>
<td>20</td>
</tr>
<tr>
<td>Command injection</td>
<td>101</td>
<td>41</td>
<td>90</td>
</tr>
<tr>
<td>Path traversal</td>
<td>169</td>
<td>19</td>
<td>80</td>
</tr>
<tr>
<td>Prototype pollution</td>
<td>192</td>
<td>126</td>
<td>158</td>
</tr>
<tr>
<td>ReDoS</td>
<td>98</td>
<td>78</td>
<td>59</td>
</tr>
<tr>
<td>Total</td>
<td>600</td>
<td>285</td>
<td>407</td>
</tr>
</tbody>
</table>
Installation and Execution

- One folder per vulnerability
  - package.json to install vulnerable package and its dependencies
  - Executable exploit as a test case
  - JSON file with meta-data
- 12 minutes to install entire benchmark
- 13 minutes to execute all exploits
Applications

- Finding mislabeled vulnerable versions
- Finding flawed fixes
- Localizing sink calls (see paper)
- Evaluate detection and mitigation techniques
Finding Vulnerable Versions

- Which **versions** of a package are **affected**?
- For each version of the vulnerable package
  - Install package in this version
  - Try to **run** exploit
Number of Vulnerable Versions

![Bar chart showing the distribution of vulnerable versions in packages. The x-axis represents the number of vulnerable versions, ranging from 1 to more than 100, and the y-axis shows the number of packages. The chart indicates that the majority of packages have between 1 and 10 vulnerable versions, with a significant peak between 11 and 20 vulnerable versions.]
Some vulnerabilities affect only a few versions

Others affect many versions (maximum: 1,487)
Mislabeled Version Ranges

- Vulnerability databases indicate range of affected versions
  - Basis, e.g., for npm’s security warnings

- Are these ranges correct?
  - 168 versions in 19 packages are incorrectly labeled as non-vulnerable
Examples

Vulnerable

<table>
<thead>
<tr>
<th>underscore. string</th>
<th>Snyk</th>
<th>SecBench.js</th>
</tr>
</thead>
<tbody>
<tr>
<td>jspdf</td>
<td>Snyk</td>
<td>SecBench.js</td>
</tr>
<tr>
<td>ts-dot-prop</td>
<td>Snyk</td>
<td>SecBench.js</td>
</tr>
<tr>
<td>changeset</td>
<td>Snyk</td>
<td>SecBench.js</td>
</tr>
<tr>
<td>shvl</td>
<td>Snyk</td>
<td>SecBench.js</td>
</tr>
</tbody>
</table>

Time
Examples

Mislabeled as non-vulnerable, but actually can be exploited!
Examples

Affects legacy versions
Affects the latest available version: Zero-day!
Finding Flawed Fixes

- Fix may **overfit to a proof-of-concept attack**
- E.g., prototype pollution
  - Can inject properties via `obj.__proto__ and obj.constructor.prototype`
- For each vulnerability
  - Update to **latest version**
  - If exploit not successful:
    - Check if **simple mutations of exploit** work
Results

■ 18 successful exploits of “fixed” versions
  - Twelve new CVEs
■ Surprisingly simple way of finding zero-day vulnerabilities
“Fixed” version of Mozilla’s `convict` package:

```javascript
const path = k.split('.
const childKey = path.pop()
const pKey = path.join('.
if (!( pKey == '__proto__' ||
    pKey == 'constructor' ||
    pKey == 'prototype'))) {
  const parent = walk(this._instance, pKey, true)
  parent[childKey] = v
}
```
“Fixed” version of Mozilla’s *convict* package:

```javascript
const path = k.split('.
) const childKey = path.pop()
const pKey = path.join('.
) if (!( pKey == '__proto__ ' ||
    pKey == 'constructor ' ||
    pKey == 'prototype ')) {
    const parent = walk(this._instance , pKey , true)
    parent[childKey] = v
}
```

Works for the original exploit, but fails to prevent writes to, e.g.,
```
constructor.prototype.x
```
Other Applications of SecBench.js

■ Evaluation of vulnerability detection techniques
  □ How many of all vulnerabilities can they find?
  □ E.g. evaluation of “Bimodal Taint Analysis” (ISSTA’23)

■ Evaluation of mitigation techniques
  □ How many of all exploits can they prevent?

■ Empirical studies
  □ Static and dynamic properties of vulnerabilities, exploits, and fixes
SecBench.js – Conclusion

- First benchmark of JavaScript vulnerabilities that is
  - Realistic
  - Executable
  - Two-sided
  - Vetted

- Side product: 20 zero-day vulnerabilities

See ICSE’23 paper and https://github.com/cristianstaicu/SecBench.js