A Sense of Time for JavaScript and Node.js

First-Class Timeouts as a Cure for Event Handler Poisoning

James C. Davis
Eric R. Williamson
Dongyoon Lee
Contributions

**Attack:** Event Handler Poisoning
- Definition
- Analysis

**Detect + recover:** First-Class Timeouts
- Concept
- Prototype

**Engagement** with the Node.js community
- Guide
- Core APIs: Documentation and repairs
Node.js: A JS framework for web services

7M+ developers (2017)  2x YoY
760K+ modules (Aug. 2018)  2x YoY
24B+ module downloads/month (July 2018)  12x YoY
Web server architectures
One Thread per Client Architecture (OTPCA)

• Each client gets its own worker thread
• Multithreading enables scalability
• Example: Apache
Event-Driven Architecture (EDA)

- Clients multiplexed; shared threads reduce threading overhead
- **Cooperative multitasking** via (1) Partitioning and (2) Offloading
- Example: node.js
Server architecture dictates programming style

**OTPCA**

Preemptive multi-tasking
Synchronous

```python
def serveFile(req):
    cont = readFile(req.file)
    z = zip(cont)
    e = encrypt(z)
    return e
```

**EDA**

Cooperative multi-tasking
Asynchronous

```python
def serveFile(req):
    cont = await readFile(req.file)
    z = await zip(cont)
    e = await encrypt(z)
    return e
```
Event Handler Poisoning Attacks (EHP)
The EDA gains **efficiency**, loses **isolation**

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Threads</th>
<th># Threads</th>
<th>Multi-tasking</th>
</tr>
</thead>
<tbody>
<tr>
<td>OTPCA</td>
<td>Dedicated</td>
<td>Thousands</td>
<td>Preemptive</td>
</tr>
<tr>
<td>EDA</td>
<td>Shared</td>
<td>Tens</td>
<td>Cooperative</td>
</tr>
</tbody>
</table>

Event Handlers = limited resource
Exhaust resource $\rightarrow$ DoS
Behavior during EHP attack on the Event Loop

- Event loop is poisoned
- Throughput drops to zero

On the worker pool: $k$ malicious requests
def serveFile(name):
    if name.match(r'/(\/.+)+$/):
        readFile(name)
        .then(
            ...
        )

IO-DoS

Arbitrary file read

ReDoS

Super-linear regex

Vulnerable server

IO-DoS

Arbitrary file read
ReDoS-based EHP attack

Inject malicious input during steady-state

Throughput (reqs/sec)

Time (seconds)

Baseline REDOS
NodeCure REDOS

TimeoutException delivered
35% of NPM vulnerabilities enable EHP

266 IO-DoS

- Directory Traversal (CWE 22)
- Cross-Site Scripting (CWE 79)
- Man in the Middle (CWE 300)
- Denial of Service (CWE 400)
- Command Injection (CWE 77)
- Malicious Package (CWE 506)
- Information Exposure (CWE 201)
- Improper Authentication (CWE...)
- Other (CWEs 330, 208, 601, 90, ...)

EHP
Not EHP

115 ReDoS

35% of NPM vulnerabilities enable EHP.
What should we do about EHP?
Naïve 1: Restart the server

**Idea**
- Heartbeat on each Event Handler
- If any heartbeats fail, restart the server

**Problems**
- Every connected client gets DoS’d
- Repeat attacks
Naïve 2: Prevent through partitioning

```python
def sum(L):
    s = 0
    for n in L:
        s += n
    return s

async def sum(L):
    s = 0
    until done:
        s += <10 numbers>
        yield
    return s
```
Partitioning is partial and ad hoc

Only protects code under the application dev.’s control

- Not **modules**
- Not **framework**
- Not **language**

Good for algorithms – but how to meaningfully partition I/O?

Ongoing maintenance burden
Our proposed solution:

*First-Class Timeouts*
First-Class Timeouts

Analogy

Buffer overflow → Out of bounds exception
EDA “time overflow” → Timeout exception

Idea

**Time-aware** cooperative multi-tasking

- Bound the *synchronous time* of every Callback and Task
- Deliver a `TimeoutException` if this bound is exceeded

Analysis

- **Soundly** defeats EHP attacks
- **Straightforward refactoring**: `try-catch` in Promise chains
- **Non-destructive**: Existing clients unharmed
Node.cure
Design and Evaluation
## Desired behavior

<table>
<thead>
<tr>
<th>Event Handler</th>
<th>Old behavior</th>
<th>New behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event Loop</td>
<td>Unbounded execution</td>
<td>Throw TimeoutException</td>
</tr>
<tr>
<td>Worker Pool</td>
<td>&quot;</td>
<td>Return TimeoutException</td>
</tr>
</tbody>
</table>
Adding first-class timeouts to Node.js

- Application
- JS Engine
- Time-aware JS
- TimeoutException
- Interrupt handler
- V8

Node.js libs
- JS
- C++

Event-ing
- Event loop
- libuv

Time-aware Worker Pool
- Managers watch for timeouts
- Disposable Workers

Timeout Watchdog
- Monitor CB entry/exit
- Set T.E. interrupt
Node.cure prototype

- **Built** on Node.js v8.8.1 (LTS)
  - 4 KLoC across 50 files
- **Compatible**
  - Passes Node.js core test suite*
- **Available** on GitHub
Security guarantees

• Every vulnerable **Language** and **Framework** API is safe
  • Applications built with these APIs are safe, too!
• Passes our EHP test suite
  • All vulnerable Node.js APIs
  • Including all used in the npm vulnerabilities

However

• **Detect**: Must choose timeout thresholds (*Goldilocks problem*)
• **Respond**: Tight threshold or blacklisting
## Performance penalty

### Micro-benchmarks

<table>
<thead>
<tr>
<th>Component</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>New interrupt</td>
<td>0%</td>
</tr>
<tr>
<td>Instr. CBs</td>
<td>1.01-2.4 x</td>
</tr>
<tr>
<td>I/O buffers</td>
<td>1.3 x</td>
</tr>
</tbody>
</table>

### Macro-benchmarks (summary)

<table>
<thead>
<tr>
<th>App. type</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server</td>
<td>0-2 %</td>
</tr>
<tr>
<td>Utility</td>
<td>0-8 %</td>
</tr>
<tr>
<td>Middleware</td>
<td>6-24 %</td>
</tr>
</tbody>
</table>
Community Engagement
Guide on nodejs.org

Reviews Node.js architecture
EHP attacks + examples
Advice about npm module safety

Don’t Block the Event Loop (or the Worker Pool)

Should you read this guide?

If you're writing anything more complicated than a brief command-line script, reading this should help you write higher-performance, more-secure applications.

This document is written with Node servers in mind, but the concepts apply to complex Node applications as well. Where OS-specific details vary, this document is Linux-centric.

TL; DR

Node.js runs JavaScript code in the Event Loop (initialization and callbacks), and offers a Worker Pool to handle expensive tasks like file I/O. Node scales well, sometimes better than
## Changes to Node.js core

<table>
<thead>
<tr>
<th>Documentation</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>readFile</td>
<td></td>
</tr>
<tr>
<td>randomBytes</td>
<td></td>
</tr>
<tr>
<td>randomFill</td>
<td></td>
</tr>
<tr>
<td>spawn</td>
<td></td>
</tr>
</tbody>
</table>
Closing Remarks
The EDA has an EHP problem.

First-class timeouts can cure it.

We:

- **Defined** an attack
- **Demonstrated** its presence in the wild
- **Designed** and **prototyped** a defense
- **Disseminated** to the practitioner community

Thank you for your attention!
Bonus Material
Choosing a timeout

• The tighter the timeout, the less effective the EHP attack
• Loose timeouts $\rightarrow$ blacklist attackers
  • No DDoS (threat model)
  • Blacklisting is relatively easy with First-Class Timeouts because the `TimeoutException` is delivered in the context of the malicious request
Choose timeout – minimize CB variance during tuning
  • Goldilocks problem

Add error handling – a global exception handler and per-request handlers

New first-class asynchronous primitives like async/await and Promises make this possible

We only support global timeouts but could refine thresholds on a per-CB and per-Task basis
Various ideas towards EHP-safety

- Heartbeat
- Partitioning
- First-class timeouts
- Larger worker pool
- Preemptible callbacks and tasks
- Speculative concurrent execution
- Serverless

Within the EDA paradigm

Dedicating resources to each client: OTPCA
Threat model

- Attacker can trigger worst-case behavior
- No DDoS

Thus:
- Include EHP, a problem unique to EDA
- Exclude DDoS, a general problem for problem web servers
More details on time-aware Event Handlers

Event Loop

Priority Executor

uv_queue_work_prio

Manager
Always has a Worker

Worker Pool

Executors

Worker
Disposable

Hangman

Dangling Worker

uv_queue_work
### Implementation details

<table>
<thead>
<tr>
<th>Layer</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language</td>
<td>• Add <code>TimeoutException</code></td>
</tr>
<tr>
<td></td>
<td>• Add interrupt</td>
</tr>
<tr>
<td>Framework</td>
<td>• Timeout Watchdog</td>
</tr>
<tr>
<td></td>
<td>• Handle T.E. from async APIs</td>
</tr>
<tr>
<td></td>
<td>• Offload sync. APIs</td>
</tr>
<tr>
<td></td>
<td>• Time-aware C++ add-ons</td>
</tr>
<tr>
<td>Application</td>
<td>• Handle T.E.</td>
</tr>
</tbody>
</table>
C++ add-ons

- Node.js applications can contain:
  - Pure JavaScript
  - C++ add-ons
    - e.g. for performance or using systems libraries

- Application-defined C++ add-ons are unprotected by F.C.T
  - Must be made time-aware, similar to how we made Node.js’s own C++ bindings time-aware
  - Only 0.7% of npm modules have C++ add-ons
Experimental slides
Node.js attack – with ReDoS and IO-DoS

![Graph showing throughput versus time with NodeCure and Baseline results]

- Throughput (req/sec)
- Time (seconds)

Legend:
- Baseline REDOS
- Baseline ReadDOS
- NodeCure REDOS
- NodeCure ReadDOS
One-thread-per-client architecture (OTPCA)

Dispatcher

Thread pool

Handle request

Handle request
Event-driven architecture (EDA)

Pending events
returns completed work

offloads

Event Loop
Worker Pool
Long-running request in OTPCA

Dispatcher

Thread pool

Handle request

☠
Long-running request in EDA

Event Loop

Worker Pool

Pending events
returns completed work

offloads