Abstract
The software development community has begun to adopt the Event-Driven Architecture (EDA) to provide scalable web services, most prominently through Node.js. Though the EDA offers excellent scalability, it comes with an inherent risk: the Event Handler Poisoning (EHP) Denial of Service attack. In essence, EHP attacks say that the scalability of the EDA is a double-edged sword: the EDA is scalable because clients share a small set of threads, but if an attacker causes these threads to block then the server becomes unusable. EHP attacks are a significant threat, as hundreds of popular Node.js-based websites were recently reported to be vulnerable to forms of EHP attacks.

In this work we make three contributions. First, we formally define EHP attacks, and we show that EHP attacks are a common form of vulnerability in the largest EDA community, the Node.js ecosystem. Second, we design and evaluate an EHP-safe defense: first-class timeouts. Although realizing first-class timeouts is difficult in a complex real-world framework like Node.js, our Node.cure prototype defends Node.js applications against all known EHP attacks. Third, we have identified and documented or corrected vulnerable APIs in Node.js, and our guide on avoiding EHP attacks is now available on nodejs.org.

Node.cure is effective, defeating all known EHP attacks with application overheads as low as 0%. More generally, we show that Node.cure offers strong security guarantees against EHP attacks for the majority of the Node.js ecosystem.

Millions of developers have embraced the EDA, but without strict adherence to the EDA paradigm their code is vulnerable to EHP attacks. Node.cure offers the cure.

1 Introduction
Web services are the lifeblood of the modern Internet. To minimize costs, service providers want to maximize the number of clients each server can handle. Over the past decade, this goal has led the software community to seriously consider a paradigm shift in their software architecture — from the One Thread Per Client Architecture (OTPCA) used in Apache to the Event-Driven Architecture (EDA) championed by Node.js.

The EDA is not a passing fad. Perhaps inspired by Welsh et al.’s SEDA concept [100], server-side EDA frameworks like Twisted [25] have been in use since at least the early 2000s. But the boom in the EDA has come with Node.js. Node.js (“server-side JavaScript”) was introduced in 2009 and is now used by many organizations in many industries, including IBM, Microsoft, Apple, Cisco, Intel, RedHat, GE, Siemens, NASA, Wikipedia, and the NFL [32, 56, 81, 76, 36, 16]. The associated package ecosystem, npm, boasts 575,000 modules [55]. It is not an exaggeration to state that Node.js is becoming a critical component of the modern web [18, 55].

Given the importance of the EDA to the modern web, it has received surprisingly little study from a security perspective [51, 78]. We present the first formal treatment of the weaknesses in the EDA, and describe a Denial of Service attack, Event Handler Poisoning (EHP), that can be used against EDA-based services like Node.js applications (§3).

EHP attacks observe that the source of the EDA’s scalability is also its Achilles’ heel. Where the OTPCA gives every client its own thread, the EDA multiplexes many clients onto a small number of Event Handlers (threads) to reduce per-client overheads. Because many clients share the same Event Handlers, an EDA-based server must correctly implement cooperative multitasking [91]. An incorrect implementation of cooperative multitasking enables an EHP attack: an attacker dominates the time spent by an Event Handler, preventing the server from handling all clients fairly. We found that although the EHP problem is little discussed in the Node.js documentation, EHP vulnerabilities are common in npm (§4).

In §5 we define criteria for EHP-safety, and use these criteria to evaluate four EHP defenses ([9]). Though variations of three of these defenses are used today, they are ad hoc and thus impractical in an ecosystem dominated by third-party modules. The fourth defense, the Timeout Approach, requires extending existing EDA languages...
and frameworks, but it provides a universal solution for EHP-safety with strong security guarantees. The Timeout Approach makes timeouts a first-class member of an EDA framework, securing both types of Event Handlers with a universal timeout mechanism.

Our Node.cure prototype (7) demonstrates the Timeout Approach in the complex Node.js framework, completely defending real applications against EHP attacks. Node.cure spans the entire Node.js stack, modifying the Node.js JavaScript engine (V8, C++), core modules (JavaScript and C++), and the EDA mechanism (libuv, C), and can secure real applications with low overhead (8). We feel our work is timely, as Staicu and Pradel recently reported that hundreds of popular websites are vulnerable to one type of EHP attack with minimal attacker effort (9). Node.cure defeats this and all other EHP attacks.

In summary:
1. We formally define Event Handler Poisoning (EHP) (3), a DoS attack against the EDA. We systematically demonstrate that EHP attacks are common in the largest EDA community, the Node.js ecosystem (4).
2. We describe a general antidote for Event Handler Poisoning attacks: first-class timeouts. We demonstrate effectiveness with our Node.cure prototype for Node.js in (7). We evaluate the security guarantees of timeouts (strong, 6) and their costs (small, 8).
3. Our findings have been corroborated by the Node.js community. Our guide on EHP-safe techniques is on nodejs.org, and our PRs documenting and improving unsafe Node.js APIs have been merged (9).

2 Background

In this section we review the EDA (2.1), explain our choice of EDA framework for study (2.2), and introduce directly related work (sections 2.3 and 2.4).

2.1 Overview of the EDA

There are two paradigms for web servers, distinguished by the ratio of clients to resources, which corresponds to an isolation-performance tradeoff. The One Thread Per Client Architecture (OTPCA) dedicates resources to each client, for strong isolation but higher memory and context-switching overheads (81). The Event-Driven Architecture (EDA) tries the opposite approach, with many clients sharing execution resources: client connections are multiplexed onto a single-threaded Event Loop, with a small Worker Pool for expensive operations.

All mainstream server-side EDA frameworks use the Asymmetric Multi-Process Event-Driven (AMPED) architecture (82). This architecture (hereafter “the EDA”) is illustrated in Figure 1. In the EDA, the OS or a framework place events in a queue, and the Callbacks of pending events are executed sequentially by the Event Loop. The Event Loop may offload expensive Tasks like file I/O to the queue of a small Worker Pool, whose Workers execute Tasks and generate “Task Done” events for the Event Loop when they finish (60). We refer to the Event Loop and the Workers as Event Handlers.

Figure 1: This is the AMPED Event-Driven Architecture. Incoming events from clients A and B are stored in the event queue, and the associated Callbacks (CBs) will be executed sequentially by the Event Loop. We will discuss B’s EHP attack (CB1), which has poisoned the Event Loop, in §3.3.

Because the Event Handlers are shared by all clients, the EDA has a particular development paradigm. Each Callback and Task is guaranteed atomicity: once scheduled, it runs to completion. Atomicity calls for cooperative multitasking (91), and developers partition the generation of responses into multiple stages. The effect of this partitioning is to regularly yield to other requests by deferring work until the next stage (52). This partitioning results in a Lifeline (59), a DAG describing the partitioned steps needed to complete an operation. A Lifeline can be seen by following the arrows in Figure 1.

2.2 Node.js among other EDA frameworks

Examples of EDA frameworks include Node.js (JavaScript) (13), libuv (C/C++) (9), Vert.x (Java) (26), Twisted (Python1) (25), and Microsoft’s F# (56). These frameworks have been used to build a wide variety of industry and open-source services (e.g. 6, 82, 65, 77, 30, 29, 7, 3).

Most prominent among these frameworks is Node.js, a server-side event-driven framework for JavaScript introduced in 2009. The popularity of Node.js comes from its promise of “full stack JavaScript” — client- and server-side developers can speak the same language and share the same libraries. This vision has driven the rise of the JavaScript package ecosystem, npm, which with 575,000 modules is the largest of any language (55). Node.js is still accelerating: use doubled between 2016 and 2017, from 3.5 million developers (31) to 7 million (33).

The Node.js codebase has three major parts (61), whose interactions complicate top-to-bottom extensions

---

1 In addition, Python 3.4 introduced native EDA support.
like Node.cure. An application’s JavaScript code is executed using Google’s V8 JavaScript engine [63], its programmability is supported by Node.js core JavaScript modules with C++ bindings for system calls, and the event-driven architecture is implemented in C using libuv [9].

2.3 Algorithmic complexity attacks

Our work is inspired by Algorithmic Complexity (AC) attacks ([74, 50]), a form of Denial of Service attack. In an AC attack, a malicious client crafts input that triggers the worst-case complexity of the victim server’s algorithms, causing execution resources to be wasted. Well-known examples of AC attacks include attacks on hash tables [50] as well as Regular expression Denial of Service (ReDoS) attacks, which exploit the worst-case exponential-time behavior typical of regular expression (regexp) engines [49].

As will be made clear in §3, EHP attacks are not simply the application of AC attacks to the EDA. The first distinction is a matter of perspective: where AC attacks focus on the complexity of the algorithms the service employs, EHP attacks target the software architecture used by the service, and are only concerned with time. The second distinction is one of definition: AC attacks rely on CPU-bound activities to achieve DoS, while EHP attacks may use either CPU-bound or I/O-bound activities to poison an Event Handler.

2.4 Preliminary work on EHP attacks

Davis et al.’s workshop paper [51] was the first to sketch EHP attacks. Their description was superficial, lacking a threat model or a formal definition. Their (unimplemented) solution was the “C-WCET Principle”, which would incur significant refactoring costs at the language, framework, and application levels.

Our work offers a formal treatment of the EHP problem ([3], and introduces criteria for EHP-safety ([5] that explain why the C-WCET Principle is both ad hoc and unsuited to dealing with I/O-based EHP attacks ([4]). We then propose and prototype the Timeout Attack in our Node.cure extension of Node.js (sections 7 and 8), and engage with the Node.js developer community to increase awareness of EHP attacks ([9]).

3 Event Handler Poisoning Attacks

In this section we provide our threat model ([3.1], formally define Event Handler Poisoning (EHP) attacks ([3.2], and present an attack taxonomy ([3.5]). To illustrate the problem, we demonstrate minimal EHP attacks using ReDoS and “ReadDoS” ([3.3].

3.1 Threat model

We assume the following. The victim is an EDA-based server, e.g. a Node.js server, with an EHP vulnerabilities. The attacker knows how to exploit this vulnerability: they know the victim feeds user input to a Vulnerable API, and they know evil input that will cause the Vulnerable API to block the Event Handler executing it. Lastly, once identified, the victim will blacklist the attacker.

3.2 A formal definition of an EHP attack

Total/synchronous complexity and time. Before we define EHP attacks, we must introduce a few definitions. First, recall the standard EDA formulation illustrated in Figure 1. Each client request is handled by a Lifeline ([39] partitioned into one or more Callbacks and Tasks. A Lifeline is a DAG whose vertices are Callbacks or Tasks and whose edges are events or Task submissions.

We define the total complexity of a Lifeline as the cumulative complexity of all of its vertices as a function of their cumulative input. The synchronous complexity of a Lifeline is the greatest individual complexity among its vertices. A Lifeline’s total time and synchronous time are defined analogously. Since the EDA relies on cooperative multitasking, a Lifeline’s synchronous complexity and synchronous time provide theoretical and practical bounds on how vulnerable it is. Note that a Lifeline with large total time is not vulnerable so long as each vertex (Callback/Task) has a small synchronous time; only synchronous time matters for EHP.

EHP attacks. An EHP attack exploits an EDA-based service with an incorrect implementation of cooperative multitasking. Here is how it works. The attacker identifies an exploitable Lifeline, one with large worst-case synchronous complexity, and poisons the corresponding large-complexity Callback or Task with evil input. This evil input causes the Event Handler executing it to block, starving pending events or Tasks.

An EHP attack can be carried out against either the Event Loop or the Workers in the Worker Pool. A poisoned Event Loop brings the server to a halt, while for each poisoned Worker the responsiveness of the Worker Pool degrades. Thus, an attacker’s aim is to poison either the Event Loop or enough of the Worker Pool to harm the throughput of the server. Remember, the Worker Pool is small enough that poisoning it will not attract the attention of network-level defenses.

Trends in software development [89] have made EHP attacks relatively easy both to find and to launch. When commercial applications embrace open-source software, including open-source package ecosystems, vulnerabilities in these (open-source) libraries may become vulnerabilities in the application. In [4.2] we show that many npm modules have exploitable EHP vulnerabilities, and that hundreds of real Node.js-based websites have been affected. Even worse, these example EHP attacks are asymmetric, requiring a few small evil inputs from an
attacker to poison an entire EDA-based server.

### 3.3 Illustrating EHP attacks: ReDoS and ReadDoS

To illustrate EHP attacks, we developed a minimal vulnerable file server with EHP vulnerabilities common in real npm modules ([4, 2]). Figure 2 shows pseudocode, with the EHP vulnerabilities indicated: ReDoS on line 2, and ReadDoS on line 3. In Figure 3 you can see the impact of EHP attacks on baseline Node.js, as well as the effectiveness of Node.cure. In a real website, these attacks would result in partial or complete DoS. Without Node.cure the only remedy would be to restart the server, dropping all existing client connections.

ReDoS attacks have been well documented in the literature ([49, 90, 96, 69, 88, 97]). A string composed of /'s followed by a newline triggers exponential-backtracking behavior in Node.js’s regular expression engine, poisoning the Event Loop in a CPU-bound EHP attack.

The second EHP vulnerability might be more surprising. Our server has a directory traversal vulnerability, permitting clients to read arbitrary files. In the EDA, directory traversal vulnerabilities can be paralyzed into I/O-bound EHP attacks, “ReadDoS”, provided the attacker can identify a Slow File from which to read. Since line 3 uses the asynchronous framework API readFile, each ReadDoS attack on this server will poison a Worker in an I/O-bound EHP attack.

The architecture-level behavior of the ReDoS attack is illustrated in Figure 1. After client A’s benign request is sanitized (CB_A1), the readFile Task goes to the Worker Pool (Task_A1), and when the read completes the Callback returns the file content to A (CB_A2). Then client B’s malicious request arrives and triggers ReDoS (CB_B1), dropping the server throughput to zero. The ReadDoS attack has a similar effect on the Worker Pool, with the same unhappy result.

### 3.4 Aren’t EHP attacks possible in the OTPCA?

EHP attacks are only possible when clients share execution resources; they are not possible when clients are isolated as in the OTPCA. In the EDA, all clients share one Event Loop and a small Worker Pool. For example, Node.js has a maximum of 128 Workers ([17]). By contrast, in the OTPCA a similar attack poisons only one of the thousands of “Event Handlers”. We believe EHP attacks have not previously been explored because the EDA has only recently seen popular adoption by the server-side community.

### 3.5 A taxonomy of EHP attacks

We turn now to a precise taxonomy of the root causes of EHP vulnerabilities. At a high level, large synchronous time can be attributed to the use of a Vulnerable API that performs computation or I/O.

Table 1 classifies Vulnerable APIs along two axes. Along the first axis, a Vulnerable API affects either the Event Loop or a Worker, and it might be CPU-bound or I/O-bound. Along the second axis, a Vulnerable API can be found in the language, the framework, or the application. In our evaluation we provide an exhaustive list of Vulnerable APIs for Node.js ([8, 1]). Although our examples are specific to Node.js, the same general classification applies to any EDA framework.

### 4 EHP Attacks In the Wild

With a definition of EHP attacks in hand, we now assess the awareness and incidence of EHP vulnerabilities in the Node.js ecosystem. We found that the Node.js documentation does not carefully outline the need for cooperative multitasking ([4, 2]), which might contribute to the fact that 26% of known npm module vulnerabilities can be used for EHP attacks ([4, 2]; indeed, some already have been [94].
### Table 1: Taxonomy of Vulnerable APIs in Node.js, with examples.

<table>
<thead>
<tr>
<th>Vuln. APIs</th>
<th>Event Loop (1.2)</th>
<th>Worker Pool (1.1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language</td>
<td>Regexp, JSON</td>
<td>N/A</td>
</tr>
<tr>
<td>Framework</td>
<td>Crypto, zlib</td>
<td>FS</td>
</tr>
<tr>
<td>Application</td>
<td>while(1)</td>
<td>DB query</td>
</tr>
</tbody>
</table>

Table 1: Taxonomy of Vulnerable APIs in Node.js, with examples. An EHP attack through a Vulnerable API poisons the Event Loop or a Worker, and its synchronous time is due to CPU-bound or I/O-bound activity. A Vulnerable API might be part of the language, framework, or application, and might be largely synchronous (Event Loop) or asynchronous (Worker Pool). zlib is the Node.js compression library. N/A: JavaScript has no native Worker Pool nor any I/O APIs.

#### 4.1 The Node.js docs are quiet

We believe that examining the documentation and guides for JavaScript and Node.js should tell us whether this community is aware of EHP attacks.

There are three places to look for Node.js documentation. First, there is the JavaScript language specification. Second, Node.js maintains documentation for the APIs of its core modules [19]. Third, the nodejs.org website offers a set of guides for new developers [15].

Although there are potential EHP vulnerabilities in an implementation of JavaScript (like ReDoS or large JSON object manipulation), language specifications do not always discuss their risks. For example, in MDN’s documentation only eval has warnings [11].

The Node.js documentation does not indicate any vulnerabilities (like ReDoS) in its JavaScript engine. However, it does give hints about avoiding EHP attacks on the Node.js framework APIs. The Node.js APIs include several modules with Vulnerable APIs, and developers are warned away from the synchronous versions. The asynchronous versions, which could poison the Worker Pool, bear the warning: “[These] APIs use libuv’s threadpool, which can have surprising and negative performance implications for some applications,” and refer to another page reading “libuv’s threadpool has a fixed size...other...APIs that run in libuv’s threadpool [may] experience degraded performance.” While technically sound, we feel that this level of documentation may be unhelpful for the average Node.js developer.

The most critical shortcoming we identified is in the Node.js new developer guides, which skip directly from “Hello world” to deep dives on HTTP and profiling. These guides do not advise developers on the design of Node.js applications, which must fit the EDA paradigm and avoid EHP vulnerabilities. Later we describe the guide we wrote on these topics [19], now available on nodejs.org.

#### 4.2 EHP vulnerabilities are common

While documentation can tell us about community awareness, vulnerability databases tell us about the state of practice. Although the community documentation does not discuss EHP attacks, we found that EHP vulnerabilities are common in npm modules, thus affecting any Node.js applications that rely on these modules.

We systematically reproduced the analysis of Davis et al. [51] on the Snyk.io vulnerability database [23]. Davis et al. used a manual categorization of the 353 npm vulnerabilities reported by Snyk.io as of February 2017. We reproduced their analysis on the 838 npm vulnerabilities reported as of January 2018, using regular expressions for a consistent classification scheme. Figure 4 shows the distribution of vulnerability types, absorbing categories with fewer than 10 vulnerabilities into Other. A high-level CWE number is given next to each class.

The dark bars in Figure 4 show the 223 vulnerabilities (26%) that can be employed in an EHP attack under our threat model (1.3). The 155 EHP-relevant Directory Traversal vulnerabilities are exploitable because they allow arbitrary file reads, which can poison the Event Loop or the Worker Pool through ReadDoS (1.3). The 59 EHP-relevant Denial of Service vulnerabilities poison the Event Loop; 48 are ReDoS, and the remaining 11 can trigger infinite loops or worst-case performance in inefficient algorithms. In Other are 9 Arbitrary File Write vulnerabilities that, like ReadDoS, can be used for EHP attacks by writing to Slow Files.

Due to the frequent practice of using untrusted third-party npm modules in production code [37], EHP vulnerabilities in npm translate directly into EHP vulnerabilities in Node.js servers. For example, Staicu and Pradel recently showed how low the bar is for attackers. After remarking that “a skilled individual can attack real-world websites with moderate effort,” they used ReDoS vulnerabilities in popular npm modules to DoS hundreds of websites from the Alexa Top Million [94].

#### 5 What Does it Mean to be EHP-Safe?

Having defined the EDA (1.2), explained EHP attacks (1.3), and surveyed their reality (1.4), we now ponder what it means to be EHP-Safe.

---

**Figure 4:** Classification of the 838 npm module vulnerabilities, by category and by usefulness in EHP attacks. Data from 29 January 2018.
The fundamental cause of EHP vulnerabilities should be clear: EHP vulnerabilities stem from Vulnerable APIs that fail to honor cooperative multitasking. As categorized in Table 1, Vulnerable APIs can poison the Event Loop or the Worker Pool, they can be CPU-bound or I/O-bound, and they occur at any level of the application stack: language (vulnerable constructs), framework (vulnerable core modules), or application (vulnerable code in the application or libraries).

To be EHP-safe, an application must use APIs whose synchronous time (§3.2) is (small and) bounded. If an application cannot bound the synchronous time of its APIs, it is vulnerable to EHP attack; conversely, if an application can bound the synchronous time of its APIs, then it is EHP-safe. This application-level guarantee must span the application’s full stack: the language constructs it uses, the framework APIs it invokes, and the application code itself.

In §5, we outline several approaches to EHP-safety. To choose between them, we suggest three criteria:

1. General. A developer must be able to use existing APIs or slight modifications thereof. This ensures that applications are general, that they can solve meaningful problems.

2. Developer-friendly. A developer should not need expertise in topics outside of their application domain.

3. Composable. If a developer uses only EHP-safe APIs, they should be unable to create a Vulnerable API. In a composable EHP-safe system, an API containing while(1); would not be Vulnerable, because it is composed only of non-Vulnerable language constructs. In contrast, without compositability a system has only ad hoc EHP safety, because in addition to making language- and framework-level APIs non-Vulnerable, all application-level APIs composed of them must also be assessed.

6 Antidotes for Event Handler Poisoning

Here we discuss four schemes by which to guarantee EHP-safety: removing Vulnerable APIs (§6.1), only calling Vulnerable APIs with safe input (§6.2), refactoring Vulnerable APIs into safe ones (§6.3), and bounding a Callback or Task’s synchronous time at runtime (§6.4). We evaluate these schemes using the criteria from §5.

6.1 Remove calls to Vulnerable APIs

To become EHP-safe, an application might eliminate calls to Vulnerable APIs. Obviously, this Blacklist Approach fails the generality criterion.

6.2 Sanitize input to Vulnerable APIs

Many Vulnerable APIs are only problematic if they are called with unsafe input. Another approach to EHP-safety, then, is the Sanitary Approach: ensure that every Vulnerable API is always called with a safe input. The Sanitary Approach is developer-unfriendly, because it requires developers to identify evil input for the APIs they call. And when Vulnerable APIs like regexps are used to filter input, quis custodiet ipsos custodes?

6.3 Refactor Vulnerable APIs

A more promising path to EHP-safety is to refactor Vulnerable APIs into safe ones, as proposed by Davis et al. [51]. Since the goal of EHP-safety is to bound the synchronous time of a Lifeline, a developer could restrict the complexity of each of its Callbacks and Tasks, perhaps to Constant Worst-Case Execution Time (C-WCET). This would bound the synchronous complexity of the Lifeline. If all Vulnerable APIs were replaced with APIs following the C-WCET Principle, the application would be EHP-safe. The C-WCET Principle is the logical extreme of cooperative multitasking, extending input sanitization from the realm of data into the realm of the algorithms used to process it.

Evaluating the C-WCET Principle. The C-WCET Principle has two benefits. First, by bounding the synchronous complexity of all Callbacks and Tasks, there would be no such thing as evil input. Every client request could be handled. Second, the C-WCET Principle can be applied largely without changing the language specification. For example, as has previously been proposed [48], a non-exponential-time regexp engine could be used in place of Node.js’s exponential-time Irregexp engine [47], and most applications would be none the wiser.

The C-WCET Principle meets the generality criterion; every API can be partitioned towards C-WCET. It is also somewhat developer-friendly: language- and framework-level APIs can be C-WCET partitioned by framework developers, leaving developers responsible for C-WCET partitioning their own APIs. One difficulty, though, is that application developers rely heavily on third-party npm modules that might not be C-WCET-partitioned.

Critically, like the Blacklist and Sanitary Approaches, the C-WCET Principle is not composable. Since Vulnerable APIs can always be composed from non-Vulnerable APIs, developers would need to take responsibility for applying and maintaining the C-WCET Principle at the application level.

It is worth noting that while the C-WCET Principle applies readily to CPU-bound algorithms, it is unclear how to achieve C-WCET Partitioning for I/O. Computation can be partitioned to the instruction granularity, but the granularity of I/O is poorly defined. For example, the C-WCET Principle would have no trouble with ReDoS.

3 As noted in §5, an API consisting of a while(1); loop is Vulnerable in the Blacklist, Sanitary, and C-WCET Approaches, even though it only calls the constant-time “language APIs” while(1) and ;.
but cannot cleanly combat ReadDoS.

6.4 Dynamically bound the running time

The goal of the C-WCET Principle was to bound a Line-
line’s synchronous complexity as a way to bound its syn-
chronous time, and would require refactoring every Vul-
nerable API. But instead of statically bounding an API’s
synchronous complexity, what if we could dynamically
bound its synchronous time? Then the worst-case com-
plexity of each Callback and Task would become irre-
relevant, because they would be unable to take more than
the quantum provided by the runtime. In this Timeout
Approach, the runtime detects and aborts long-running
Callbacks and Tasks by emitting a TimeoutError, thrown
from Callbacks and returned from Tasks.

The Timeout Approach says that in the EDA, pure co-
operative multitasking is unsafe. Just as safe general-
purpose languages offer null pointer exceptions and
buffer overflow exceptions, so a safe EDA framework
must offer timeout exceptions, and developers must be
prepared to deal with them. Developers should not have
to explicitly wrap sensitive code in timers or offload it
to separate threads or processes; in the EDA, timeouts
should be provided by the framework just like any other
security-relevant exception.

The Timeout Approach provides EHP-safety. Like the
C-WCET Principle, the Timeout Approach is general. It
is more developer-friendly because it only requires de-
velopers to handle TimeoutErrors in their own APIs.
And where the C-WCET Principle was ad hoc, the Time-
out Approach is composable. If the EDA runtime had a
sense of time, then a long-running Callback/Task could
be interrupted no matter where it was, whether in the lan-
guage, the framework, or the application.

Although the Timeout Approach fulfills all of the
EHP-safety criteria, like C-WCET Partitioning it re-
quires application-level refactoring. Because the pro-
posed TimeoutErrors emitted by the runtime are not cur-
rently part of the language and framework specification,
existing applications would need to be ported. We do not
anticipate this to be a heavy burden for well-written ap-
lications, whose Callbacks should already be exception-
aware to catch errors like null pointers (ReferenceError)
and buffer overflows (RangeError), and which should
test the result of Tasks for errors.

Though the C-WCET Principle and the Timeout Ap-
proach both meet the generality criterion, the Timeout
Approach is more developer-friendly, and is the only
EDA-centric solution we analyzed that is composable.7

Objections. One objection to the Timeout Approach
is that “timeouts are hardly novel.” We disagree. To
the best of our knowledge, existing timeouts take one of
two approaches. The first is on a per-API basis: the
.NET framework has used timeouts to combat ReDoS
since 2012 [20]. Unfortunately, such ad hoc timeouts
will always be non-composable. The second approach
is on a per-thread basis: OSes commonly use a heart-
beat mechanism to detect and restart unresponsive ap-
lications, and in the OTPCA a client thread can easily
be killed and replaced if it exceeds a timeout. But the
same approach fails in the EDA. Detecting and restart-
ing a blocked Event Loop will break all existing client
connections, achieving the very DoS we seek to defeat.
Remember the double-edged sword of the EDA: scal-
ability at the cost of client isolation. Because clients are
not isolated on separate execution resources, our Time-
out Approach makes timeouts a first-class member of
JavaScript and Node.js, non-destructively guaranteeing
that no Event Handler can block on an event or Task.

Another objection to the Timeout Approach is that
“someone must select a timeout,” and there is some evi-
dence that little thought has been given to the choice of
timeouts [83]. For the purpose of avoiding EHP attacks
under our threat model, however, this does not pose a se-
rious problem. Remember that a typical web server han-
dles hundreds or thousands of clients per second (Fig-
ure 3). If so, simple arithmetic tells us that in a Node.js
server, individual Callbacks and Tasks must be taking no
longer than milliseconds to complete. Thus, a univer-
sal Callback-Task timeout on the order of 1 second will
discomfit normal Callbacks and Tasks, will certainly
ensure that an EHP attack is detected and defeated expe-
diently, and might briefly annoy existing clients but not
significantly undermine their experience. It then falls to
the application to blacklist the attacker.

The last objection is “it’s trivial!” We report, however,
that though the Timeout Approach is conceptually sim-
ple, realizing it in a complex real-world framework like
Node.js is difficult. The challenges that face a sound im-
plementation are:

1. Enforcing timeouts on both Callbacks and Tasks.
2. Enforcing timeouts in both the language (V8) and the
framework (Node.js core modules and libuv).
3. Ensuring that timeouts are low-overhead.

7 First-Class Timeouts in Node.cure

Our Node.cure prototype implements the Timeout Ap-
proach for Node.js by bounding the synchronous time of
every Callback and Task. To do so, Node.cure makes
timeouts a first-class member of Node.js, extending the
JavaScript specification by introducing a TimeoutError
to abort long-running Callbacks and Tasks.

At a high level, here is the timeout behavior
Node.cure enforces. A long-running Callback poisons
the Event Loop; in Node.cure these Callbacks throw a
TimeoutError whether they are in JavaScript or in a

\footnote{We discuss alternative approaches to achieve EHP-safety in Appendix A.}
Node.js C++ binding ([7.2]). A long-running Task poisons its Worker; in Node.cure, such a Task is aborted and fulfilled with a TimeoutError ([7.1]).

As a result, Node.cure meets the composability criterion of [5] the Node.js language and framework APIs are timeout-aware, and an application whose APIs are composed of these APIs will be EHP-safe ([7.3]). Node.cure also has an exploratory Slow Resource Policy to reduce timeout overheads ([7.4]).

We describe the Timeout Approach for Tasks first, because it introduces machinery used to apply the Timeout Approach to Callbacks.

7.1 Timeout-aware Tasks

EHP attacks targeting the Worker Pool use Vulnerable APIs to submit long-running Tasks that poison a Worker. Node.cure defends against such attacks by bounding the synchronous time of Tasks.

Timeout-aware Worker Pool. We modified the libuv Worker Pool to be timeout-aware, replacing libuv’s Workers with Executors that combine a permanent Manager with a disposable Worker. If a Task times out, the Manager uses a Hangman to dispose of the poisoned Task and its Worker, and creates a new Worker to resume handling Tasks. These roles are illustrated in Figure 5.

Here is a slightly more detailed description of our timeout-aware Worker Pool. Node.js’s Worker Pool is implemented in libuv, exposing a Task submission API uv_queue_work, which we extended as shown in Table 2. The original Workers of the Worker Pool would simply invoke work and then notify the Event Loop to invoke done. This is also the typical behavior of our timeout-aware Workers. When a Task takes too long, however, the potentially-poisoned Worker’s Manager invokes the Task’s timed_out callback. If the submitter does not request an extension, the Manager creates a replacement Worker so that it can continue to process subsequent Tasks, creates a Hangman thread for the poisoned Worker, and notifies the Event Loop that the Task timed out. The Event Loop then invokes its done Callback with a TimeoutError, permitting a rapid response to evil input. Concurrently, once the Hangman successfully kills the Worker thread, it invokes the Task’s killed callback for resource cleanup, and returns.

Differentiating between timed_out and killed permits more flexible error handling, but introduces technical challenges. If a rapid response to a timeout is unnecessary, then it is simple to defer done until killed finishes. If a rapid response is necessary, then done must be able to run before killed finishes, resulting in a Dangling Worker problem: an API’s work implementation may access externally-visible state after the Event Loop receives the associated TimeoutError. We addressed the Dangling Worker problem in Node.js’s Worker Pool customers using a mix of killed Waiting, message passing, and blacklisting.

<table>
<thead>
<tr>
<th>Callback</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void work</td>
<td>Perform Task.</td>
</tr>
<tr>
<td>int timed_out*</td>
<td>When Task has timed out. Can request extension.</td>
</tr>
<tr>
<td>void done</td>
<td>When Task is done. Special error code for timeout.</td>
</tr>
<tr>
<td>void killed*</td>
<td>When a timed_out Task’s thread has been killed.</td>
</tr>
</tbody>
</table>

Table 2: Summary of the Worker Pool API. work is invoked on the Worker. done is invoked on the Event Loop. The new callbacks, timed_out and killed, are invoked on the Manager and the Hangman, respectively. On a timeout, work, timed_out, and done are invoked, in that order; there is no ordering between the done and killed callbacks, which sometimes requires reference counting for safe memory cleanup. *New callbacks.

Timeout-aware Worker Pool Customers. The Node.js APIs affected by this change (i.e. those that create Tasks) are in the encryption, compression, DNS, and file system modules. In all cases we allowed timeouts to proceed, killing the long-running Worker. Handling encryption and compression was straightforward, while the DNS and file system APIs were more complex.

Node.js’s asynchronous encryption and compression APIs are implemented in Node.js C++ bindings by invoking APIs from openssl and zlib, respectively. If the Worker Pool notifies these APIs of a timeout, they wait for the Worker to be killed before returning, to ensure it no longer modifies state in these “black-box” libraries nor accesses memory that might be released after done is invoked. Since openssl and zlib are purely computational, the Dangling Worker is killed immediately.

Node.js implements its file system and DNS APIs by relying on libuv’s file system and DNS support, which on Linux make the appropriate calls to libc. Because the libuv file system and DNS implementations share memory between the Worker and the submitter, we modified them to use message passing for memory safety of Dangling Workers — wherever the original implementation’s work accessed memory owned by the submitter, e.g. for
read and write, we introduced a private buffer for work and added copyin/copyout steps. In addition, we used `pthread_setcancelstate` to ensure that Workers will not be killed while in a non-cancellable libc API [5]. DNS is read-only so there is no risk of the Dangling Worker modifying external state. In the file system, write modifies external state, but we avoid any Dangling Worker state pollution via blacklisting. Our blacklisting-based Slow Resource policy is discussed in more detail in [7.4].

At the top of the Node.js stack, when the Event Loop sees that a Task timed out, it invokes the application’s Callback with a `TimeoutError`.

7.2 Timeouts for Callbacks

Node.cure defends against EHP attacks that target the Event Loop by bounding the synchronous time of Callbacks. To make Callbacks timeout-aware, we introduce a TimeoutWatchdog that monitors the start and end of each Callback and ensures that no Callback exceeds the timeout threshold. We time out JavaScript instructions using V8’s interrupt mechanism ([7.2.1]), and we modify Node.js’s C++ bindings to ensure that Callbacks that enter these bindings will also be timed out ([7.2.2]).

7.2.1 Timeouts for JavaScript

**TimeoutWatchdog**. Our TimeoutWatchdog instruments every Callback using the experimental Node.js async-hooks module [14], which allows an application to register special Callbacks before and after a Callback is invoked.

Before a Callback begins, our TimeoutWatchdog starts a timer. If the Callback completes before the timer expires (“good path”), we erase the timer. If the timer expires, the watchdog signals V8 to interrupt JavaScript execution by throwing a `TimeoutError`. The watchdog then starts another timer, ensuring that timeouts while handling the previous `TimeoutError` are also detected.

Although a precise TimeoutWatchdog can be implemented to carefully honor the timeout threshold, the resulting communication overhead between the Event Loop and the TimeoutWatchdog can be non-trivial ([8.2]). A cheaper alternative is a lazy TimeoutWatchdog, which simply wakes up at intervals of the timeout threshold and checks whether the Callback it last observed is still executing; if so, it emits a `TimeoutError`. A lazy TimeoutWatchdog reduces the overhead of making a Callback, but decreases the precision of the `TimeoutError` threshold.

**V8 interruptions**. To handle the TimeoutWatchdog’s request for a `TimeoutError`, Node.cure extends the interrupt infrastructure of Node.js’s V8 JavaScript engine [63] to support timeouts. In V8, low priority interruptions like a pending garbage collection are checked regularly (e.g. each loop iteration, function call, etc.), but no earlier than after the current JavaScript instruction finishes. High priority interrupts take effect immediately, interrupting long-running JavaScript instructions. Timeouts require the use of a high priority interrupt because they must be able to interrupt long-running individual JavaScript instructions like `str.match(regexp)` (possible ReDoS).

To support a `TimeoutError`, we modified V8 as follows: (1) We added the definition of a `TimeoutError` into the Error class hierarchy; (2) We added a `TimeoutInterrupt` into the list of high-priority interrupts; and (3) We added a V8 API to raise a `TimeoutInterrupt`. The TimeoutWatchdog calls this API, which interrupts the current JavaScript stack by throwing a `TimeoutError`.

The only JavaScript instructions that V8 instruments to be interruptible are regexp matching and JSON parsing; these are the language-level Vulnerable APIs. Other JavaScript instructions are viewed as effectively constant-time, so these interrupts are evaluated e.g. at the end of basic blocks. We agreed with the V8 developers in this [5] and did not attempt to instrument other JavaScript instructions to poll for pending interrupts.

7.2.2 Timeouts for the Node.js C++ bindings

The TimeoutWatchdog described in [7.2.1] will interrupt any Vulnerable APIs implemented in JavaScript, including language-level APIs like regexp and application-level APIs that contain blocking code like `while(1);`. It remains to give a sense of time to the Node.js C++ bindings that allow the JavaScript code in Node.js applications to interface with the broader world.

Node.js has asynchronous and synchronous C++ bindings. The asynchronous bindings are safe because they do a fixed amount of work synchronously to submit a Task and then return. However, the synchronous C++ bindings complete the entire operation on the Event Loop before returning, and therefore must be given a sense of time. The relevant Vulnerable synchronous APIs are those in the file system, cryptography, and compression modules. The `execSync` API is also Vulnerable, but is only intended for scripting purposes.

Because the Event Loop holds the state of all pending clients, we cannot use the Hangman `pthread_cancel` approach, as this would result in the DoS the attacker desired. Another alternative is to offload the request to the Worker Pool and await its completion, but this would incur high request latencies when the Worker Pool’s queue is not empty. Instead, we extended the Worker Pool paradigm with a dedicated `Priority Executor` whose queue is exposed via a new API:

---

5For example, we found that string operations complete in milliseconds even when a string is hundreds of MBs long.
uv_queue_work_prio (Figure 5). This Executor follows the same Manager-Worker-Hangman paradigm as the Executors in Node.cure’s Worker Pool. To make these Vulnerable synchronous APIs timeout-aware, we offload them to the Priority Executor using the existing asynchronous implementation of the API, and had the Event Loop await the result. Because these synchronous APIs are performed on the Event Loop as part of a Callback, we propagate the Callback’s remaining time to this Executor’s Manager to ensure that the TimeoutWatchdog’s timer is honored.

7.3 Timeouts for application-level Vulnerable APIs

Let us revisit Node.cure’s composability guarantee. As described above, Node.cure makes Tasks (§7.1) and Callbacks (§7.2) timeout-aware to defeat EHP attacks against language and framework APIs. The Timeout Approach is composable, so an application composed of calls to these APIs will be EHP-safe.

However, applications can still escape the reach of the Timeout Approach by defining their own C++ bindings. These bindings would need to be made timeout-aware, following the example we set while making Node.js’s Vulnerable C++ bindings timeout-aware (file system, DNS, encryption, and compression). Without refactoring, applications with their own C++ bindings may not be EHP-safe. In our evaluation we found that application-defined C++ bindings are rare (§8.3).

7.4 Exploring policies for slow resources

Our Node.cure runtime detects and aborts long-running Callbacks and Tasks executing on Node.js’s Event Handlers. For unique evil input this is the best we can do, as accurately predicting whether a not-yet-seen input will time out is difficult. If an attacker might re-use the same evil input multiple times, however, we can track whether or not an input led to a timeout and short-circuit subsequent requests that use this input with an early timeout.

While evil input memoization could in principle be applied to any API, the size of the input space to track is a limiting factor. The evil inputs that trigger CPU-bound EHP attacks like ReDoS exploit properties of the vulnerable algorithm and are thus usually not unique. In contrast, the evil inputs that trigger I/O-bound EHP attacks like ReadDoS must name a particularly slow resource, presenting an opportunity to short-circuit requests on this slow resource.

In Node.cure we implemented a slow resource management policy for libuv’s file system APIs, targeting those that reference a single resource (e.g. open, read, write). When one of the APIs we manage times out, we mark the file descriptor and the associated inode number as slow. We took the simple approach of permanently blacklisting these aliases by aborting subsequent accesses\(^6\) with the happy side effect of solving the Dangling Worker problem for write. This policy is appropriate for the file system, where access times are not likely to change.\(^7\) For DNS queries, timeouts are likely due to a network hiccup, and a temporary blacklist might be more appropriate.

7.5 Prototype details

Node.cure is built on top of Node.js LTS v8.8.1,\(^7\) a recent long-term support version of Node.js. Our prototype is for Linux, and added 4,000 lines of C, C++, and JavaScript code across 50 files spanning V8, libuv, the Node.js C++ bindings, and the Node.js JavaScript libraries.

Node.cure is a complete and correct implementation of the Timeout Approach. It passes the core Node.js test suite, with a handful of failures due to bad interactions with experimental or deprecated features. In addition, several cases fail when they invoke rarely-used file system APIs we did not make timeout-aware. Real applications run on Node.cure without difficulty (Table 3).

In Node.cure, timeouts for Callbacks and Tasks are controlled by environment variables. Our implementation would readily accommodate a fine-grained assignment of timeouts for individual Callbacks and Tasks.

8 Evaluating Node.cure

We evaluated Node.cure in terms of its effectiveness (§8.1), costs (§8.2), and security guarantees (§8.3). In summary: with a lazy TimeoutWatchdog, Node.cure defeats all known EHP attacks with low overhead on the “good path”, estimated at 1.3x using micro-benchmarks and manifesting at 1.0x-1.24x using real applications. Node.cure guarantees EHP-safety to all Node.js applications that do not define their own C++ bindings, a rare practice.

All measurements provided in this section were obtained on an otherwise-idle desktop running Ubuntu 16.04.1 (Linux 4.8.0-56-generic), 16GB RAM, Intel i7 @3.60GHz, 4 physical cores with 2 threads per core. For a baseline we used Node.js LTS v8.8.1 from which Node.cure was derived, compiled with the same flags. We used a default Worker Pool (4 Workers).

8.1 Effectiveness

To evaluate the effectiveness of Node.cure, we developed an EHP test suite that makes EHP attacks of all types shown in Table 6. Our suite is comprehensive and conducts EHP attacks using every Vulnerable API we...
identified, including the language level (regexp, JSON), framework level (all Vulnerable APIs from the file system, DNS, cryptography, and compression modules), and application level (infinite loops, long string operations, array sorting, etc.). This test suite includes each type of real EHP attack from our study of EHP vulnerabilities in npm modules (§4.2). Node.cure defeats all 92 EHP attacks in this suite: each synchronous Vulnerable API throws a TimeoutError, and each asynchronous Vulnerable API returns a TimeoutError.

To finish our suite, we ported a vulnerable npm module to Node.cure. We selected the node-oniguruma module, which offers asynchronous regexp APIs. Since the Oniguruma regexp engine is vulnerable to ReDoS, its APIs are Vulnerable. However, they evaluate the regexps in a Task rather than in a Callback, and so poison a Worker rather than the Event Loop. We made node-oniguruma timeout-aware by using our modified Worker Pool uv_queue_work API (Table 2). Node.cure times out ReDoS attacks against this module’s Vulnerable APIs.

8.2 Costs

The Timeout Approach introduces runtime overheads, which we evaluated using micro-benchmarks and macro-benchmarks. It may also require some refactoring, and a choice of timeout.

**Overhead: Micro-benchmarks.** Whether or not they time out, Node.cure introduces several sources of overheads to monitor Callbacks and Tasks. We evaluated the following overheads using micro-benchmarks:

1. Every time V8 checks for interrupts, it now tests for a pending timeout as well.
2. Both the lazy and precise versions of the TimeoutWatchdog instrument every asynchronous Callback using async-hooks, with relative overhead dependent on the complexity of the Callback.
3. To ensure memory safety for Dangling Workers, Workers operate on buffered data that must be allocated when the Task is submitted. For example, Workers must copy the I/O buffers supplied to read and write twice.

**New V8 interrupt.** We found that the overhead of our V8 Timeout interrupt was negligible, simply a test for one more interrupt in V8’s interrupt infrastructure.

**TimeoutWatchdog’s async hooks.** We measured the additional cost of invoking a Callback due to TimeoutWatchdog’s async hooks. A lazy TimeoutWatchdog increases the cost of invoking a Callback by 2.4x, and a precise TimeoutWatchdog increases the cost by 7.9x. Of course, this overhead is most noticeable in Callbacks that are empty. As the number of instructions in a Callback increases, the cost of the hooks amortizes. For example, if the Callback executes 500 empty loop iterations, the lazy overhead drops to 1.3x and the precise overhead drops to 2.7x; at 10,000 empty loop iterations, the lazy and precise overheads are 1.01x and 1.15x, respectively.

**Worker buffering.** Our timeout-aware Worker Pool requires buffering data to accommodate Dangling Workers, affecting DNS queries and file system I/O. While this overhead will vary from API to API, our micro-benchmark indicated a 1.3x overhead using read and write calls with a 64KB buffer.

**Overhead: Macro-benchmarks.** Our micro-benchmarks suggested that the overhead introduced by Node.cure may vary widely depending on what an application is doing. Applications that make little use of the Worker Pool will only pay the overhead of the additional V8 interrupt (minimal) and the TimeoutWatchdog’s async hooks, whose cost is strongly dependent on the size of the Callbacks. Applications that use the Worker Pool will also pay the overhead of Worker buffering (variable, perhaps 1.3x).

We chose macro-benchmarks using a GitHub pot-pourri technique: we searched GitHub for “language:JavaScript”, sorted by “Most starred”, and identified server-side projects from the first 50 results. To add additional complete servers, we also included LokijS [10], a popular Node.js-based key-value store, and IBM’s Acme-Air airline simulation [2].

Table 3 lists the macro-benchmarks we used and the performance overhead for each type of TimeoutWatchdog. These results show that Node.cure introduces minimal overhead on real server applications, and they confirm the value of a lazy TimeoutWatchdog. Matching our macro-benchmark assessment of the TimeoutWatchdog’s async-hooks overhead, the overhead from Node.cure increased as the complexity of the Callbacks used in the macro-benchmarks decreased — the middleware benchmarks sometimes used empty Callbacks to handle client requests. In non-empty Callbacks like those of the real servers, this overhead is amortized.

**Refactoring.** Node.cure adds a TimeoutError to the JavaScript and Node.js specifications. See §9.

**Choice of timeout.** See §6.4 and §9.

8.3 Security Guarantees

We discussed the general security guarantees of the Timeout Approach in §6.4. Here we discuss the precise security guarantees of our Node.cure prototype. As described in §7, we gave a sense of time to JavaScript (including interrupting long-running regexp and JSON operations) as well as to framework APIs identified by the Node.js developers as long-running (file system, DNS, cryptography, and compression).

Because the Timeout Approach is composable, application-level APIs composed of these timeout-aware language and framework APIs are also timeout-aware.
Table 3: Results of our macro-benchmark evaluation of Node.cure’s overhead. Where available, we used the benchmarks defined by the project itself. Otherwise, we ran its test suite. Overheads are reported as “lazy, precise”, and are the ratio of Node.cure’s performance to that of the baseline Node.js, averaged over several steady-state runs. We report the average overhead because we observed no more than 3% standard deviation in all but LokiJS, which averaged 8% standard deviation across our samples of its sub-benchmarks. * Median of sub-benchmark overheads.

However, Node.js also permits applications to add their own C++ bindings, and these will not be timeout-aware without refactoring.

To evaluate the extent of this limitation, we measured how frequently developers use C++ bindings. To this end, we examined a sample of npm modules. We npm install’d 125,173 modules (about 20% of the ecosystem) using node v8.9.1 (npm v5.5.1), resulting in 23,070 successful installations. Of these, 695 (3%) had C++ bindings.

From this we conclude that the use of user-defined C++ bindings is uncommon in npm modules, unsurprising since many developers in this community come from client-side JavaScript and may not have strong C++ skills. This implies, but does not directly indicate, that C++ bindings are unusual in Node.js applications. We conclude that the need to refactor C++ bindings to be timeout-aware would not place a significant burden on the community. As a template, we performed several such refactorings ourselves in the Node.js framework and our port of Node-oniguruma.

9 Discussion

What should the EDA community do without timeouts? Although they lack first-class timeouts, developers in the EDA community must still be concerned about EHP attacks. We believe their best approach is the C-WCET Principle.[14] First, we submitted a PR with a guide to building EHP-safe EDA-based applications. It was merged and can be found on nodejs.org, where new Node.js developers go to learn best practices for Node.js development. We believe that it will give developers insights into secure Node.js programming practices.

Second, we studied the Node.js implementation and identified several unnecessarily Vulnerable APIs in Node.js (fs.readFile, crypto.randomFill, and crypto.randomBytes, all of which submit a single un-partitioned Task). Our PRs warning developers about the risks of EHP attacks using these APIs have been merged. We also submitted a trial pull request repairing the simplest of these, by partitioning fs.readFile. It was then merged after a multi-month discussion on the performance-security tradeoff involved.

What might timeout-aware programs look like? As discussed in [7] applying the Timeout Approach while preserving the EDA changes the language and framework specifications. In particular, all synchronous APIs may now throw a TimeoutError, and all asynchronous APIs may now be fulfilled with a TimeoutError. These changes have two implications for application developers: Timeout-aware applications must be able to tolerate arbitrary TimeoutErrors (just like any other exception), and developers should pursue low-variance Callbacks and Tasks (to permit the choice of a tighter timeout threshold).

Are there other avenues toward EHP-safety? In [6] we discussed several EHP-safe designs. There are of course other approaches, like significantly increasing the size of the Worker Pool, performing speculative concurrent execution [45], or using explicitly preemptable Callbacks and Tasks. However, each of these is a variation on the same theme: dedicating resources to every client, i.e. the One Thread Per Client Architecture. If the server community wishes to use the EDA, which offers high responsiveness and scalability through the use of cooperative multitasking, we believe the Timeout Approach is a good path to EHP-safety.

10 Related Work

Throughout the paper we have referred to specific related work. Here we place our work in its broader context, situated at the intersection of the Denial of Service literature and the Event-Driven Architecture community.

Denial of Service attacks. Research on DoS can be broadly divided into network-level attacks (e.g. distributed DoS attacks) and application-level attacks [38]. Since EHP attacks exploit the semantics of the application, they are application-level attacks, not easily defeated by network-level defenses.

DoS attacks seek to exhaust the resources critical to the proper operation of a server, and various kinds of exhaustion have been considered. The brunt of the literature has focused on exhausting the CPU. Some have examined the gap between common-case and worst-case performance in algorithms and data structures, for example in quicksort [74], hashing [50], and exponentially backtracking algorithms like ReDoS [49, 90] and rule matching [92]. Other CPU-centric DoS attacks have targeted more general programming issues, showing how to trigger infinite recursion [46] and how to trigger or detect infinite loops. But the CPU is not the only vulnerable resource, as shown by Olivo et al.’s work polluting databases to increase the cost of queries [79]. We are not aware of prior research work that incurs DoS using the file system, as do our ReadDoS attacks, though we have found a handful of CVE reports to this effect [13].

Our work identifies and shows how to exploit the most limited resource of the EDA: Event Handlers. To launch a similar attack in the OTPCA, attackers must submit enough queries to overwhelm the OS’s preemptive scheduler, no mean feat. In contrast, in the EDA there are only a few Event Handlers to be poisoned. Although we prove our point using previously-reported attacks like ReDoS, the underlying resource we are exhausting is not the CPU but the small, fixed-size set of Event Handlers deployed in EDA-based services.

Security in the EDA. Though researchers have studied security vulnerabilities in different EDA frameworks, most prominently client-side applications, the security properties of the EDA itself have not been investigated in detail.

Our work is most closely related to the workshop paper of Davis et al. [51], discussed earlier ([2.4]). Less immediately relevant is the Node.js-specific high-level survey of Ojamaa et al. [78], which mentioned the rule “Don’t block the event loop”, and Stuicu and Pradel’s study on code injection vulnerabilities in npm [95], which can be used for EHP attacks. DeGroef et al. [52]’s proposed method to securely integrate third-party modules from npm is not effective for EHP attacks because Node.js lacks a sense of time.

More broadly, other security research in the EDA has studied client-side JavaScript/Web [70, 68, 53, 75] and Java/Android [58, 57, 41, 67] applications. These have often focused on platform-specific issues like DOM issues in web browsers [70].

EHP attacks. The server-side EDA practitioner community is aware of the risk of DoS due to (synchronous) EHP on the Event Loop. A common rule of thumb is “Don’t block the Event Loop”, advised by many tutorials as well as recent books about EDA programming for Node.js [98, 44]. Wandschneider suggests worst-case linear-time partitioning on the Event Loop [98], while Casciaro advises developers to partition any computation on the Event Loop, and to offload computationally expensive tasks to the Worker Pool [44]. Our work offers a more rigorous taxonomy and evaluation of EHP attacks, and in particular we extend the rule of “Don’t block the Event Loop” to the Worker Pool.

EHP attacks on the server side may correspond to performance bugs in client-side EDA programs. Liu et al. studied such bugs [72], and Lin et al. proposed an automatic refactoring to offload expensive tasks from the Event Loop to the Worker Pool [71]. This approach is sound on a client-side system like Android, where the demand for threads is limited, but is not applicable to the server-side EDA because the small Worker Pool is shared among all clients. For the server-side EDA, Lin et al.’s approach would need to be extended to incorporate automatic C-WCET partitioning.

As future work, we believe that research into computational complexity estimation ([80, 65, 86]) and measurement ([87, 62, 43]) might be adapted to the Node.js context for EHP vulnerability detection and automatic refactoring using the C-WCET Principle. This may be difficult because of the complexity of statically analyzing JavaScript and the EDA ([73, 40, 64, 99, 59]).

11 Conclusion

The Event-Driven Architecture (EDA) holds great promise for scalable web services, and it is increasingly popular in the software development community. In this paper we formally defined Event Handler Poisoning attacks, which exploit the cooperative multitasking at the heart of the EDA. The Node.js community has endorsed our expression of this problem, hosting our guide at nodejs.org.

We proposed several defenses against EHP attacks, and prototyped the most promising: first-class timeouts. Our prototype, Node.cure, defends Node.js applications against all known EHP attacks. It is available on GitHub [13].

Our findings can be directly applied by the EDA community, and we hope they influence the design of existing and future EDA frameworks. With the rise of popularity in the EDA, EHP attacks will become an increasingly critical DoS vector, a vector that can be defeated with first-class timeouts.


13We omit the reference to meet the double-blind requirement.
Acknowledgments

We thank the CCS, NDSS, S&P, and USENIX Security referees for their helpful feedback, as well as XXX for their excellent shepherding. L. Cheng, C. Coglan, A. Kazerouni, S. Rahaman, J. Terner, and the Virginia Tech Systems Reading Group were helpful sounding boards for our ideas and manuscripts, as were M. Hicks, G. Wang, and D. Yao. J.D. Greef of Ronomon suggested several of the EHP attacks listed in the discussion.

References

[36] How cond nast used node.js to unify brands and create customer faith. https://www2.thelinuxfoundation.org/node-casestudies-condenast


[53] DE GROEF, W., DEVRIESE, D., NIKIFORAKIS, N., AND PIESSENS, F. Flowfox: A web browser with flexible and precise information flow control. CCS.


[76] O’DELL, J. Exclusive: How LinkedIn used Node.js and HTML5 to build a better, faster app, 2011.

[77] O’DELL, J. Exclusive: How LinkedIn used Node.js and HTML5 to build a better, faster app, 2011.


