A Sense of Time for JavaScript and Node.js
First-Class Timeouts as a Cure for Event Handler Poisoning

James C. Davis
Virginia Tech
Eric R. Williamson
Virginia Tech
Dongyoon Lee
Virginia Tech

Abstract
The software development community is adopting the Event-Driven Architecture (EDA) to provide scalable web services, most prominently through Node.js. Though the EDA scales well, it comes with an inherent risk: the Event Handler Poisoning (EHP) Denial of Service attack. When an EDA-based server multiplexes many clients onto few threads, a blocked thread (EHP) renders the server unresponsive. EHP attacks are a serious threat, with hundreds of vulnerabilities already reported in the wild.

We make three contributions against EHP attacks. First, we describe EHP attacks, and show that they are a common form of vulnerability in the largest EDA community, the Node.js ecosystem. Second, we design a defense against EHP attacks, First-Class Timeouts, which incorporates timeouts at the EDA framework level. Our Node.cure prototype defends Node.js applications against all known EHP attacks with overheads as low as 0%. Third, we promote EHP awareness in the Node.js community. We analyzed Node.js for vulnerable APIs and documented or corrected them, and our guide on avoiding EHP attacks is available on nodejs.org.

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 consider shifting from the One Thread Per Client Architecture (OTPCA) used in Apache to the Event-Driven Architecture (EDA) championed by Node.js.

Perhaps inspired by Welsh et al.’s SEDA concept [97], server-side EDA frameworks like Twisted [23] 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 widely used in industry, including at IBM [56], Microsoft [22], Paypal [66], eBay [81], LinkedIn [76], and others [30] [15] [35]. Node.js’s package ecosystem, npm, boasts over 625,000 modules [55]. Node.js is becoming a critical component of the modern web [17] [34].

In this paper we 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 a two-edged sword. While the OTPCA gives every client its own thread at the cost of context-switching overheads, 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 fair cooperative multitasking [89]. Otherwise an EHP attack is born: an attacker’s request can unfairly dominate the time spent by an Event Handler, preventing the server from handling other clients. We report that EHP vulnerabilities are common in npm modules [34].

We analyze two approaches to EHP-safety in §4 and propose First-Class Timeouts as a universal defense with strong security guarantees. Since time is a precious resource in the EDA, built-in TimeoutErrors are a natural mechanism to protect it. Just as OutOfBoundsErrors allow applications to detect and react to buffer overflow attacks, so TimeoutErrors allow EDA-based applications to detect and react to EHP attacks.

Our Node.cure prototype [5] implements First-Class Timeouts in the Node.js framework. First-Class Timeouts require changes across the entire Node.js stack, from the language runtime (V8), to the event-driven library (libuv), to the core libraries. Our prototype secures real applications from all known EHP attacks with low overhead [6].

Our findings have been corroborated by the Node.js community [17]. We have developed a guide for practitioners on building EHP-proof systems, updated the Node.js documentation to warn developers about the perils of several APIs, and improved the safety of the fs.readFile API.

In summary, here are our contributions:
1. We analyze the denial-of-service potential inherent in the event-driven architecture. We define Event Han-
**Publisher's Note**

This is a draft version of the article. The final version may differ slightly.

---

**2 Background**

In this section we review the EDA (§2.1), explain our choice of EDA framework for study (§2.2), and describe relevant prior work (§2.3).

### 2.1 Overview of the EDA

There are two paradigms for web servers, distinguished by the ratio of clients to resources. The One Thread Per Client Architecture (OTPCA) dedicates resources to each client, for strong isolation but higher memory and context-switching overheads [83]. 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 [83]. This architecture (hereafter “the EDA”) is illustrated in Figure 1. In the EDA, the OS or a framework places 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 [59]. We refer to the Event Loop and the Workers as Event Handlers.

Because the Event Handlers are shared by all clients, the EDA requires a particular development paradigm. Each Callback and Task is guaranteed atomicity: once scheduled, it runs to completion on its Event Handler. Because of the atomicity guarantee, if an Event Handler blocks, the time it spends being blocked is wasted rather than being preempted. Without preemptive multitasking, developers must implement cooperative multitasking to avoid starvation [89]. They do this by partitioning the handling of each client request into multiple stages, typically at I/O boundaries. For example, with reference to Figure 1 a Callback might perform some string operations in $CB_{A1}$, then offload a file I/O to the Worker Pool in $Task_{A1}$ so that another client’s request can be handled on the Event Loop. The result of this partitioning is a per-request Lifeline [42], 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

There are many EDA frameworks, including Node.js (JavaScript) [13], libuv (C/C++) [9], Vert.x (Java) [24], Twisted (Python) [25], and Microsoft’s P# [50]. These frameworks have been used to build a wide variety of industry and open-source services (e.g. [6, 82, 66, 77, 28, 27, 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 Node.js-JavaScript package ecosystem, npm, which with over 625,000 modules is the largest of any language [55]. The Node.js Foundation reported that the number of Node.js developers doubled from 3.5 million to 7 million between 2016 and 2017 [29, 51].

The Node.js framework has three major parts [61], whose interactions complicate top-to-bottom extensions like Node.cure. An application’s JavaScript code is executed using Google’s V8 JavaScript engine [63], the event-driven architecture is implemented using libuv [9], and Node.js has core JavaScript libraries with C++ bindings for system calls.

### 2.3 Algorithmic complexity attacks

Our work is inspired by Algorithmic Complexity (AC) attacks ([74] [51]), a form of Denial of Service attack. In

---

1. In addition, Python 3.4 introduced native EDA support.
an AC attack, a malicious client crafts input that shifts the performance of the victim service’s data structures and algorithms from average-case to worst-case, reducing throughput to yield DoS. Well-known examples of AC attacks include attacks on hash tables [51] and regular expressions (ReDoS) [50].

As will be made clear in §3, EHP attacks are not simply the application of AC attacks to the EDA. AC attacks focus on the complexity of the algorithms a service employs, while EHP attacks are concerned with the effect of malicious input on the software architecture used by a service. Because EHP attacks are only concerned with time, AC attacks are just one mechanism by which an EHP attack can be realized; any time-consuming operation, like I/O, is a potential EHP vector. However, not all AC attacks can be used to launch an EHP attack.

3 Event Handler Poisoning Attacks

In this section we provide our threat model (§3.1) and define Event Handler Poisoning (EHP) attacks (§3.2). In §3.3 we give two examples of EHP attacks, one CPU-bound (ReDoS) and one I/O-bound (“ReadDoS”). Lastly we show that EHP vulnerabilities are common in the modules in the npm registry.

3.1 Threat model

The victim is an EDA-based server with an EHP vulnerability. 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.

Not all DoS attacks are EHP attacks. An EHP attack must cause an Event Handler to block. This blocking could be due to computation or I/O, so long as it takes the Event Handler a long time to handle. Other ways to trigger DoS, like crashing the server through unhandled exceptions or memory exhaustion, are not time oriented and are thus out of scope. DDoS attacks are also out of scope; they consume a server’s resources with myriad light clients providing normal input, rather than one heavy client providing malicious input.

3.2 Definition of an EHP attack

Supporting definitions. Before we can define EHP attacks, we must introduce a few definitions. First, recall the standard EDA formulation illustrated in Figure 1. As discussed in §2.1, a client request is handled by a Lifeline [42], a sequence of operations 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, and depends on the degree of partitioning of the operations performed on input. Computational complexity is an appropriate measure for compute-bound vertices, but time may be more appropriate for vertices that perform I/O. Consequently, we define a Lifeline’s total time and synchronous time analogously.

If there is a difference between a Lifeline’s average and worst-case synchronous complexity (time), then we call this a vulnerable Lifeline. We attribute the root cause of the difference between average and worst-case performance to a Vulnerable API invoked in the problematic vertex.

The notion of a “Vulnerable API” is a convenient abstraction. The trouble may of course not be an API at all but the use of an unsafe language feature (e.g. ReDoS). And if an API is asynchronous, it is itself partitioned and will have its own sub-Lifeline. In this case we are concerned about the costs of those vertices.

EHP attacks. An EHP attack exploits an EDA-based service with an incorrect implementation of cooperative multitasking. The attacker identifies a vulnerable Lifeline (server API) and poisons the Event Handler that executes the corresponding large-complexity Callback or Task with evil input. This evil input causes the Event Handler executing it to block, starving pending requests.

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 the throughput of the Worker Pool will degrade for each simultaneously poisoned Worker. 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. Based on typical Worker Pool sizes, we assume the Worker Pool is small enough that poisoning it will not attract the attention of network-level defenses.

Since the EDA relies on cooperative multitasking, a Lifeline’s synchronous complexity (time) provide theoretical and practical bounds on how vulnerable it is. Note that a Lifeline with large total complexity (time) is not vulnerable so long as each vertex (Callback/Task) has small synchronous complexity (time). It is for this reason that not all AC attacks can be used for EHP attacks. If an AC attack triggers large total complexity (time) but not large synchronous complexity (time) then it is not a danger. For example, an AC attack could result in a Lifeline with \( O(n^2) \) Callbacks each costing \( O(1) \). Although many concurrent AC attacks of this form would degrade the service’s throughput, we consider DDoS out of scope.

Speaking more broadly, EHP attacks are only possible when clients share execution resources. In the OTPCA, a blocked client affects only its own thread, and frame-

\[2\] Differences in complexity are well defined. For differences in I/O time we are referring to performance outliers.
works like Apache support thousands of “Event Handlers” (client threads) [60]. In the EDA, all clients share one Event Loop and a limited Worker Pool; for example, in Node.js the Worker Pool can contain at most 128 Workers [16]. Exhausting the set of Event Handlers in the OTPCA requires a DDoS attack, while exhausting them in the EDA is trivial if an EHP vector can be found.

### 3.3 Example EHP attacks: ReDoS and ReadDoS

To illustrate EHP attacks, we developed a minimal vulnerable file server with EHP vulnerabilities common in real npm modules as described in [3,4]. Figure 2 shows pseudocode, with the EHP vulnerabilities indicated: ReDoS on line 2, and ReadDoS on line 3.

```
def serveFile(name):
    if name.match('/(^.+.)+$/): # ReDoS
        data = await readFile(name) # ReadDoS
    client.write(data)
```

The regular expression on line 2 is vulnerable to ReDoS. A string composed of /ʼs followed by a newline takes exponential time to evaluate in Node.js’s regular expression engine, poisoning the Event Loop in a CPU-bound EHP attack.

The second EHP vulnerability is on line 3. Our server has a directory traversal vulnerability, permitting clients to read arbitrary files. In the EDA, directory traversal vulnerabilities can be parlayed 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.

In Figure 3 you can see the impact of EHP attacks on baseline Node.js, as well as the effectiveness of our `Node.cure` prototype. The methodology is described in the caption. On baseline Node.js these attacks result in complete DoS, zero throughput. Without `Node.cure` the only remedy would be to restart the server, dropping all existing client connections. Unfortunately, restarting the server would not solve the problem, since the attacker could simply submit another malicious request. With `Node.cure` the server can return to its steady-state performance.

The architecture-level behavior of the ReDoS attack is illustrated in Figure 1. After client A’s benign request is sanitized (CB1), the `readFile` Task goes to the Worker Pool (Task3), and when the read completes the Callback

---

3In addition to files exposed on network file systems, `/dev/random` is a good example of a Slow File: “[r]eads from /dev/random may block” [33].
with inconsistent CWE IDs and descriptions of different classes of vulnerabilities. Based on its title, we assigned each vulnerability to one of 17 main categories based on those used by Snyk.io. We used regular expressions to ensure our classification was consistent. We iteratively improved our regular expressions until we could automatically classify 95% of the vulnerabilities, and marked the remaining 5% as “Other”.

Some of these vulnerabilities could be used to launch EHP attacks: Directory Traversal vulnerabilities that permit arbitrary file reads, Denial of Service vulnerabilities (those that are CPU-bound, e.g. ReDoS), and Arbitrary File Write vulnerabilities. We identified such vulnerabilities using regular expressions on the descriptions of the vulnerabilities in the database, manually verifying the results. In cases where the database description was too terse, we manually categorized vulnerabilities based on the issue and patch in the module’s GitHub repository.

Results. Figure 4 shows the distribution of vulnerability types, absorbing categories with fewer than 10 vulnerabilities into the “Other” category. 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 (§3.1). The 155 EHP-relevant Directory Traversal vulnerabilities are exploitable because they allow arbitrary file reads; Denial of Service vulnerabilities (those that are CPU-bound, e.g. ReDoS); and Arbitrary File Write vulnerabilities. We identified such vulnerabilities using regular expressions on the descriptions of the vulnerabilities in the database, manually verifying the results. In cases where the database description was too terse, we manually categorized vulnerabilities based on the issue and patch in the module’s GitHub repository.

4 The number of ReDoS vulnerabilities may be skewed by recent work [92].

4 Defending Against EHP Attacks

EHP vulnerabilities stem from Vulnerable APIs that fail to provide fair cooperative multitasking. If a service cannot provide a (small) bound on the synchronous time of its APIs, then it is vulnerable to EHP attacks. Conversely, if an application can bound the synchronous time of its APIs, then it is EHP-safe.

An EHP attack has two faces: mechanism (Vulnerable API) and effect (poisoned Event Handler). Thus there are two ways to defeat an EHP attack. Either the Vulnerable API can be refactored, or a poisoned Event Handler can be detected and addressed. In this section we summarize both of these approaches and then evaluate them. A similar analysis relying on manual classification appeared in our previous work [52].

4.1 Prevent through partitioning

An API is Vulnerable if there is a difference between its average-case and worst-case costs, and of course if this worst-case cost is unbearable. A service can achieve EHP safety by statically bounding the cost of each of its APIs, both those that it invokes and those that it defines itself. For example, a developer could partition every API into a sequence of Constant Worst-Case Execution Time stages. Such a partitioning would render the service immune to EHP attacks since it would bound the synchronous complexity and time of each Lifeline.

4.2 Detect and react through timeouts

The goal of the partitioning approach is to bound a Lifeline’s synchronous complexity as a way to bound its synchronous time. Instead of statically bounding an API’s synchronous complexity through program refactoring, using timeouts we can dynamically bound its synchronous time. Then the worst-case complexity of each Callback and Task would be irrelevant, because they would be unable to take more than the quantum provided by the runtime. In this approach, the runtime detects and aborts long-running Callbacks and Tasks by emitting a TimeoutError, thrown from synchronous code (Callbacks) and returned from asynchronous code (Tasks).

We refer to this approach as First-Class Timeouts and we believe it is novel. To the best of our knowledge, existing timeout schemes take one of two forms. Some are per-API, e.g. the timeout option on the .NET framework’s regex API to combat ReDoS [18]. Per-API timeouts are ad hoc by definition. The other class of timeouts is on a per-process or per-thread basis. For example, OSes commonly use a heartbeat mechanism to detect and restart unresponsive applications, and in the OTPCA a client thread can easily be killed and replaced if it exceeds a timeout. This approach fails in the EDA because clients are not isolated on separate execution resources. Detecting and restarting a blocked Event Loop will break...
all existing client connections, resulting in DoS. Because of this, timeouts must be a first-class member of an EDA framework, non-destructively guaranteeing that no Event Handler can block.

### 4.3 Analysis

**Soundness.** The partitioning approach can prevent EHP attacks that exploit high-complexity operations. However, soundly preventing EHP attacks by this means is difficult since it requires case-by-case changes. In addition, it is not clear how to apply the partitioning approach to I/O. At the application level, I/O can be partitioned at the byte granularity, but an I/O may be just as slow for 1 byte as for 1 MB. If an OS offers truly asynchronous I/O interfaces then these provide an avenue to more fine-grained partitioning, but unfortunately Linux’s asynchronous I/O mechanisms are incomplete for both file I/O and DNS resolution.

If timeouts are applied systematically across the software stack (application, framework, language), then they offer a strong guarantee against EHP attacks. When a timeout is detected, the application can respond appropriately to it. The difficulty with timeouts is choosing a threshold $\text{[85]}$, since a too-generous value still permits an attacker to disrupt legitimate requests. As a result, if the timeout threshold cannot be tightly defined, then it must be used in combination with a blacklist.

**Refactoring cost.** Both of these approaches incur a refactoring cost. For partitioning the cost is prohibitive. Up front, any APIs invoked by an EHP-safe service must have (small) bounded synchronous time. Developers would need to re-implement any third-party APIs with undesirable performance, which would be particularly problematic in a module-dominated ecosystem like Node.js. As the composition of safe APIs may be Vulnerable APIs to submit long-running Tasks that poison a Worker. Here is the desired behavior of First-Class Timeouts.

In the remainder of the paper we describe our design, implementation, and evaluation of First-Class Timeouts in Node.js. We devote a large portion of our discussion to the choice of timeout and the refactoring implications of First-Class Timeouts.

### 5 Node.cure: First-Class Timeouts for Node.js

Though First-Class Timeouts are conceptually simple, realizing them in a real-world framework like Node.js is difficult. For soundness, every aspect of the Node.js framework must be able to emit TimeoutErrors without compromising the system state, from the language to the libraries to the application logic, and in both synchronous and asynchronous aspects. For performance, monitoring for timeouts must be lightweight.

To ensure soundness, we begin with a taxonomy of the places where Vulnerable APIs can be found in a Node.js application ($\S$ 5.1). The subsequent subsections describe how we provide TimeoutErrors across this taxonomy for the Worker Pool ($\S$ 5.2) and the Event Loop ($\S$ 5.3). We discuss performance optimizations in $\S$ 5.5 and summarize our prototype in $\S$ 5.6.

#### 5.1 Taxonomy of Vulnerable APIs

Table 1 classifies Vulnerable APIs along three axes. Along the first two axes, a Vulnerable API affects either the Event Loop or a Worker, and it might be CPU-bound or I/O-bound. Along the third 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 ($\S$ 5.6). Although our examples are specific to Node.js, the same general classification applies to any EDA framework.

Although Table 1 is specific to Node.js, the same general classification can be applied to other EDA frameworks. For example, regular expressions in most mainstream languages can lead to ReDoSs. Similarly, unless care is taken, file system or DNS requests to slow resources will poison Event Handlers in many EDA frameworks.

#### 5.2 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
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. We do not consider memory accesses as I/O.

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.

synchronous time of Tasks. Node.js short-circuits long-running Tasks with a TimeoutError.

Timeout-aware Worker Pool. Node.js’s Worker Pool is implemented in libuv. As illustrated in Figure 1, the Workers pop Tasks from a shared queue and return the results to the Event Loop. Each Worker handles its Tasks synchronously.

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. Every time a Worker picks up a Task, it notifies its Manager. If the Task takes the Worker too long, the Manager kills it with a Hangman and creates a new Worker. The long-running Task is returned to the Event Loop with a TimeoutError for processing, while the new Worker resumes handling Tasks. These roles are illustrated in Figure 5.

This design required several changes to the libuv Worker Pool API. libuv exposes a Task submission API uv_queue_work, which we extended as shown in Table 2. Workers invoke work, which is a function pointer describing the Task. On completion the Event Loop invokes 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 new 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. We used synchronization primitives to prevent races when a Task completes just after it is declared timed out.

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, since they run on separate threads. 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.

Affected APIs. The Node.js APIs affected by this change (viz. 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 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-cancelable 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 §5.5.

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.

5.3 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 (§5.3.1), and we modify Node.js’s C++ bindings to ensure that Callbacks that enter these bindings will also be timed out (§5.3.2).

5.3.1 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 recursive timeouts while handling the previous TimeoutError are also detected.

V8 interrupts. 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 interrupts 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 regular expression matching and JSON parsing; these are the language-level Vulnerable APIs. Other JavaScript instructions are viewed as effectively constant-time, so these interrupts may be slightly deferred, e.g. to the end of the nearest basic block. We agreed with the V8 developers that this and did not attempt to instrument other JavaScript instructions to poll for pending interrupts.

5.3.2 Timeouts for the Node.js C++ bindings

The TimeoutWatchdog described in §5.3.1 will interrupt any Vulnerable APIs implemented in JavaScript, including language-level APIs like regular expressions 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 synchronous work to submit a Task and then return; the Tasks are protected as discussed earlier. 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 pthread_cancel it as we do poisoned Workers, since this would result in the DoS the attacker desired. We could build off of our timeout-aware Worker Pool by offloading the request to the Worker Pool and awaiting its completion, but this would incur high request latencies when the Worker Pool’s queue is not empty. We opted to combine these approaches by offloading these requests to a dedicated Worker.

In our implementation, we extended the Worker Pool paradigm with a Priority Executor whose queue is exposed via a new API: uv_queue_work.prio (Figure 3). This Executor follows the same Manager-Worker-

---

6For example, we found that string operations complete in milliseconds even when a string is hundreds of MBs long.
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.

5.4 Timeouts for application-level Vulnerable APIs
As described above, Node.cure makes Tasks ([5.2] and Callbacks ([5.3]) timeout-aware to defeat EHP attacks against language and framework APIs. An application composed of calls to these APIs will be EHP-safe.

However, an application could still escape the reach of these timeouts by defining its 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 ([6.3]).

5.5 Performance optimizations
Since First-Class Timeouts are an always-on mechanism, it is important that their performance impact be negligible. Here we describe two optimizations.

Lazy TimeoutWatchdog. Promptly detecting TimeoutErrors with a precise TimeoutWatchdog can be expensive, because the Event Loop must synchronize with the TimeoutWatchdog every time a Callback is entered and exited. If the application workload contains many small Callbacks, whose cost is comparable to this synchronization cost, then the overhead of a precise TimeoutWatchdog may be considerable.

If the timeout threshold is soft, then the overhead from a TimeoutWatchdog can be reduced by making the Event Loop-TimeoutWatchdog communication asynchronous. When entering and exiting a Callback the Event Loop can simply increment a shared counter. A lazy TimeoutWatchdog wakes up at intervals and checks whether the Callback it last observed has been executing for more than the timeout threshold; if so, it emits a TimeoutError. A lazy TimeoutWatchdog reduces the overhead of making a Callback, but decreases the precision of the TimeoutError threshold.

Slow resource policies. 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 at runtime, 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[7] 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[8]. We did not implement a policy for DNS queries. In the context of DNS, timeouts might be due to a network hiccup, and a temporary blacklist might be more appropriate.

5.6 Implementation
Node.cure is built on top of Node.js LTS v8.8.1, a recent long-term support version of Node.js[9]. 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 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[5]).

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.

6 Evaluating Node.cure
We evaluated Node.cure in terms of its effectiveness ([6.1]), costs ([6.2]), and security guarantees ([6.3]). In summary: with a lazy TimeoutWatchdog, Node.cure

---

7To avoid fd leaks, we do not eagerly abort close.
8Of course, if the slow resource is in a networked file system like NFS or GPFS, slowness might be due to a network hiccup, and incorporating temporary device-level blacklisting might be more appropriate.
9Specifically, we built Node.cure on Node.js v8.8.1 commit dc6bbb44da from Oct. 25, 2017.
detects all known EHP attacks with low overhead on the “good path”, ranging from 1.3x-7.9x on micro-benchmarks but 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.

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).

### 6.1 Effectiveness

To evaluate the effectiveness of `Node.cure`, we developed an EHP test suite that makes every type of EHP attack, as enumerated in Table 1. Our suite is comprehensive and conducts EHP attacks using every Vulnerable API we identified, including the language level (regular expressions, 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 (§3.4). `Node.cure` detects all 92 EHP attacks in this suite: each synchronous Vulnerable API throws a `TimeoutError`, and each asynchronous Vulnerable API returns a `TimeoutError`. Our suite could be used to evaluate alternative defenses against EHP attacks.

To evaluate any difficulties in porting real-world Node.js software to `Node.cure`, we ported the `node-oniguruma` [11] npm module. This module offloads worst-case exponential regular expression queries from the Event Loop to the Worker Pool using a C++ add-on. We ported it using the API described in Table 2 without difficulty, as we did for the core modules, and `Node.cure` then successfully detected ReDoS attacks against this module’s Vulnerable APIs.

### 6.2 Runtime overhead

We evaluated the runtime overhead using micro-benchmarks and macro-benchmarks. We address other costs in the Discussion.

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

1. Every time V8 checks for interrupts, it now tests for a pending timeout as well.
2. Both the precise and lazy versions of the `TimeoutWatchdog` require instrumenting 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 precise `TimeoutWatchdog` increases the cost of invoking a `Callback` by 7.9x due to the synchronous communication between `Event Loop` and `TimeoutWatchdog`, while a lazy `TimeoutWatchdog` increases the cost by 2.4x thanks to the reduced cost of asynchronous communication. While these overheads are large, note that they are for an empty `Callback`. As the number of instructions in a `Callback` increases, the cost of executing the `Callback` will begin to dominate the cost of issuing the `Callback`. For example, if the `Callback` executes 500 empty loop iterations, the precise overhead drops to 2.7x and the lazy overhead drops to 1.3x. At 10,000 empty loop iterations, the precise and lazy overheads are 1.15x and 1.01x, respectively.

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

**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 pay the overhead of the additional V8 interrupt check (minimal) and the `TimeoutWatchdog`’s async hooks, whose cost is strongly dependent on the number of instructions executed in the Callbacks. Applications that use the Worker Pool will pay these as well as the overhead of Worker buffering (variable, perhaps 1.3x).

We chose macro-benchmarks using a GitHub potpourri 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 Lok-iJS [10], a popular Node.js-based key-value store, and IBM’s Acme-Air airline simulation [2], which is used in the Node.js benchmark suite.

Table 3 lists the macro-benchmarks we used and the performance overhead for each type of `TimeoutWatchdog`. These results show that `Node.cure` introduces min-
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 “precise, lazy”, 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.

6.3 Security Guarantees

As described in [5] our Node.cure prototype implements First-Class Timeouts for Node.js. Node.cure enforces timeouts for all vulnerable JavaScript and framework APIs identified by both us and the Node.js developers as long-running: regular expressions, JSON, file system, DNS, cryptography, and compression. Application-level APIs composed of these timeout-aware language and framework APIs are also timeout-aware.

However, Node.js also permits applications to add their own C++ bindings, and these may not be timeout-aware without refactoring. To evaluate the extent of this limitation, we measured the number of npm modules that define C++ bindings. These modules typically depend on the node-gyp and/or nan modules [37, 53]. We obtained the dependency list for each of the 628,863 npm modules and found that 4,384 out of 628,863 modules (0.7%) had these dependencies.10

As only 0.7% of npm modules define C++ bindings, we conclude that C++ bindings are not widely used and that they thus do not represent a serious limitation of our approach. We found the refactoring process for C++ bindings straightforward when we performed it on the Node.js framework and the node-oniguruma module as described earlier.

7 Practitioner Community Impact

In conjunction with the development of our Node.cure prototype, we took a two-pronged approach to reach out to the EDA practitioner community. First, we published a guide on safe service architecture for Node.js on nodejs.org. Second, we studied unnecessarily Vulnerable Node.js APIs and added documentation or increased the security of these APIs.

7.1 Guide on safe service architecture

Without First-Class Timeouts, developers in the EDA community must resort to partitioning as a preventive measure. Do new Node.js developers know this? We expect they would learn from the Node.js community’s guides for new developers, hosted on the nodejs.org website. These guides skip directly from “Hello world” to deep dives on HTTP and profiling. They do not advise developers on the design of Node.js applications, which as we have discussed must fit the EDA paradigm and avoid EHP vulnerabilities.

We prepared a guide to building EHP-safe EDA-based applications, including discussions about appropriate work patterns and the risks of high-complexity operations. The pull request with the guide was merged after discussion with the community. It can be found at https://nodejs.org/en/docs/guides/dont-block-the-event-loop/. We believe that it will give developers insights into secure Node.js programming practices, and should reduce the incidence of EHP vulnerabilities in practice.

7.2 Changes to API and documentation

We studied the Node.js implementation and identified several unnecessarily Vulnerable APIs in Node.js v8. Each of fs.readFile, crypto.randomFill, and crypto.randomBytes submits a unpartitioned Task to the Worker Pool, and in each of these cases a large Task could be expensive in terms of I/O or computation. Were a careless developer to submit a large request to one of these APIs, it could cause one of the Workers to block. This risk was not mentioned in the API documentation. These APIs could instead be automatically partitioned by the framework to avoid their use as an EHP vector.

We took two steps to address this state of affairs. First, we submitted PRs warning developers against submitting large requests to these APIs, e.g. “The asynchronous version of crypto.randomBytes() is carried out in a single threadpool request. To minimize threadpool task length...

---

10The package.json files containing dependencies can be obtained from https://skimdb.npmjs.com. We counted those that matched the regex "nan|node-gyp" on 11 May 2018.
variation, partition large randomBytes requests when doing so as part of fulfilling a client request" [9]. These PRs were merged without much comment. Second, we submitted a patch improving the simplest of these APIs, fs.readFile. This API previously read the entire file in a single read request. Our patch partitions it into a series of 64KB reads. As discussed earlier, partitioning I/O is an imperfect solution, but it is better than none.

8 Discussion

Other examples of EHP attacks. Several other EHP attacks suggest themselves. First, if the EDA framework uses a garbage collected language for the Event Loop (as do Node.js, Vert.x, Twisted, etc.), then triggering many memory allocations could lead to unpredictable blockage of the Event Loop. We are not aware of any attacks of this form, but such an attack would defeat First-Class Timeouts unless the GC were partitioned. Second, Linux lacks kernel support for asynchronous DNS requests, so they are typically implemented in EDA frameworks in the Worker Pool. If an attacker controls a DNS nameserver configured as a tarpit [72] and can convince an EDA-based victim to resolve name requests using this server, then each such request will poison one of the Workers in the Worker Pool. First-Class Timeouts will protect against this class of attacks as it does ReadDoS.

Programming with First-Class Timeouts. What would it be like to develop software for an EDA framework with First-Class Timeouts? First-Class Timeouts change the language and framework specifications. First, developers must choose a timeout threshold. Then, exception-handling code will be required for both asynchronous APIs, which may be fulfilled with a TimeoutError, and synchronous APIs, which may throw a TimeoutError.

The choice of a timeout is a Goldilocks problem. Too short, and legitimate requests will result in an erroneous TimeoutError (false positive). Too long, and malicious requests will waste a lot of service time before being detected (false negative). Timeouts in other contexts have been shown to be selected without much apparent consideration [55], but for First-Class Timeouts we suggest that a good choice is relatively easy. Consider that a typical web server can handle hundreds or thousands of clients per second. Since each of these clients requires the invocation of at least one Callback on the Event Loop, simple arithmetic tells us that in an EDA-based server, individual Callbacks and Tasks must take no longer than milliseconds to complete. Thus, a universal Callback-Task timeout on the order of 1 second should not result in erroneous timeouts during the normal execution of Callbacks and Tasks, but would permit relatively rapid detection of and response to an EHP attack

By definition, First-Class Timeouts preclude the possibility of undetected EHP attacks (false negatives) with a reasonable choice of timeout, and our Node.cure prototype demonstrates that this guarantee can be provided in practice.

Developers can assign tighter timeout thresholds to reduce the impact of an EHP attack. If a tight timeout can be assigned, then a malicious request trying to trigger EHP will get about the same amount of server time as a legitimate request will, before the malicious request is detected and aborted with a TimeoutError. The lower the variance in Callback and Task times, the more tightly the timeout thresholds can be set without false positives. Though our implementation uses coarse-grained timeouts for Callbacks and Tasks, more fine-grained timeouts are possible, perhaps via an API like process.runWithTimeout(func). Appropriate coarse or fine-grained timeouts values could also be suggested automatically or tuned over the process lifetime of the server.

If a tight timeout cannot be assigned, perhaps because there is significant natural variation in the cost of handling legitimate requests, then we recommend that the TimeoutError exception handling logic incorporate a blacklist. With a blacklist, the total time wasted by EHP attacks is equal to the number of attacks multiplied by the timeout threshold. Since DDoS is outside of our threat model, this value should be small and EHP attacks should not prove overly disruptive.

After choosing a timeout, developers would need to modify their code to handle TimeoutErrors. For asynchronous APIs that submit Tasks to the Worker Pool, a TimeoutError will be delivered just like any other error, and error handling logic should already be present. This logic could be extended, for example to blacklist the client. For synchronous APIs or synchronous links in an asynchronous sequence of Callbacks, we acknowledge that it is a bit strange that an unexceptional-looking sequence of code like a loop can now throw an error, and wrapping every function with a try-catch block seems inelegant. Happily, recent trends in asynchronous programming techniques have made it easy for developers to handle these errors. The ECMAScript 6 specification made Promises a native JavaScript feature, simplifying data-flow programming (explicit encoding of a Lifeline) [44]. Promises chain permit catch-all handling of exceptions thrown from any link in the chain, so existing catch-all handlers can be extended to handle a TimeoutError.

Detecting EHP attacks without First-Class Time-
outs. Without First-Class Timeouts, a service that is not perfectly partitioned may have EHP vulnerabilities. In existing EDA frameworks there is no way to elegantly detect and recover from an EHP attack. Introducing a heartbeat mechanism into the service would enable the detection of an EHP attack, it is true, but what then? If more than one client is connected, as is inevitable given the multiplexing philosophy of the EDA, it is not feasible to interrupt the hung request without disrupting the other clients, nor it does seem straightforward to identify which client was responsible. In contrast, First-Class Timeouts will produce a TimeoutError at some point during the handling of the malicious request, permitting exception handling logic to easily respond, by dropping the client and perhaps adding them to a blacklist.

Other avenues toward EHP-safety. In §4 we described two ways to achieve EHP-safety within the existing EDA paradigm. Other approaches are also viable but they depart from the EDA paradigm. Significantly increasing the size of the Worker Pool, performing speculative concurrent execution [48], or switching to preemptable Callbacks and Tasks could each prevent or reduce the impact of EHP attacks. However, each of these is a variation on the same theme: dedicating isolated execution resources to each client, a road that leads to the One Thread Per Client Architecture. The recent development of serverless architectures [69] is yet another form of the OTPCA, with the load balancing role played by a vendor rather than the service provider. If the server community wishes to use the EDA, which offers high responsiveness and scalability through the use of cooperative multitasking, we believe First-Class Timeouts are a good path to EHP-safety.

Generalizability. Our First-Class Timeouts technique can be applied to any EDA framework. Callbacks must be made interruptible, and Tasks must be made abortable. While these properties are more readily obtained in an interpreted language, they could in principle be enforced in compiled or VM-based languages as well.

9 Related Work

JavaScript and Node.js. Ojamaa and Duuna assessed the security risks in Node.js applications [78]. Their analysis included ReDoS and other expensive computations as a means of blocking the event loop, though they overlooked the risks of I/O and the fact that the small Worker Pool makes worker pool poisoning possible.

Our preliminary work [52] sketched EHP attacks and advocated Constant Worst-Case Execution Time (CWCT) partitioning as a solution. However, analysis in the present work reports that this approach imposes significant refactoring costs and is an ad hoc security mechanism (5.3).

Other works have identified the use of untrusted third-party modules as a common liability in Node.js applications. DeGroef et al. proposed a reference monitor approach to securely integrate third-party modules from npm [54]. Vasilakis et al. went a step further in their BreakApp system, providing strong isolation guarantees at module boundaries with dynamic policy enforcement at runtime [95]. The BreakApp approach is complete enough that it can be used to defeat EHP attacks, through what might be called Second-Class Timeouts. Our work mistrusts particular instructions and permits the delivery of TimeoutErrors at arbitrary points in sequential code, while these reference monitor approaches mistrust modules and thus only permit the delivery of TimeoutErrors at module boundaries. In addition, moving modules to separate processes in order to handle EHP attacks incurs significant performance overheads at start-up and larger performance overheads than Node.cure at run-time, and places more responsibility on developers to understand implementation details in their dependencies [95].

Static analysis can be used to identify a number of vulnerabilities in JavaScript and Node.js applications. Guarneri and Livshits demonstrate static analyses to eliminate the use of vulnerable language features or program behaviors in the client-side context [64]. Staicu et al. offer static analyses and dynamic policy enforcement to prevent command injection vulnerabilities in Node.js applications [93]. Static taint analysis for JavaScript, as proposed by Tripp et al., enables the detection of other injection attacks as well [24]. The techniques in these works can detect the possibility of EHP attacks that exploit known-Vulnerable APIs (e.g. I/O like fs.readFile), but not those exploiting arbitrary computation. Our First-Class Timeouts approach is instead a dynamic detect-and-respond defense against EHP attacks.

More broadly, other research on the EDA has studied client-side JavaScript/Web [70, 68, 53, 75] and Java/Android [58, 57, 43, 67, 71] applications. These have often focused on platform-specific issues like DOM issues in web browsers [70].

Embedded systems. Time is precious in embedded systems as well. Lyons et al. proposed the use of TimeoutErrors in mixed-criticality systems to permit higher-priority tasks to interrupt lower-priority tasks [73]. Their approach incorporates timeouts as a notification mechanism for processes that have overrun their time slices, toying with preemption in a non-preemptive operating system. Our work is similar in principle but differs significantly in execution.

Denial of Service attacks. Research on DoS can be broadly divided into network-level attacks (e.g. distributed DoS attacks) and application-level attacks [41]. Since EHP attacks exploit the semantics of the appli-
cation, 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, e.g. via worst-case performance [74][51][50][90][29], infinite recursion [49], and infinite loops [91][45]. 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.

Our work identifies and shows how to exploit and protect the most limited resource of the EDA: Event Handlers. 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.

Practitioner awareness. The server-side EDA practitioner community is aware of the risk of DoS due to (asynchronous) Event Handler Poisoning (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 [96][47]. Wandschneider suggests worst-case linear-time partitioning on the Event Loop [96], while Casciaro advises developers to partition any computation on the Event Loop, and to offload computationally expensive tasks to the Worker Pool [47]. Our work offers a more complete evaluation of EHP attacks, and in particular we extend the rule of “Don’t block the Event Loop” to the Worker Pool.

Future work. Automatically identifying modules with computationally expensive paths would permit detecting EHP vulnerabilities in advance. As future work, we believe that research into computational complexity estimation ([80][65][86]) and measurement ([87][62][46]) might be adapted to the Node.js context for EHP vulnerability detection.

10 Reproducibility

Everything needed to reproduce our results is available at https://github.com/VTLeeLab/node-cure; scripts for our analysis of the Snyk.io vulnerability database, links to our contributions to the Node.js community, and the source code for the Node.cure prototype.

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 defined Event Handler Poisoning (EHP) attacks, which exploit the cooperative multitasking at the heart of the EDA. We showed that EHP attacks occur in practice already, and as the EDA rises in popularity we believe that EHP attacks will become an increasingly critical DoS vector. The Node.js community has endorsed our expression of this problem, hosting our guide to avoiding EHP attacks on nodejs.org.

We proposed two defenses against EHP attacks, and prototyped the more promising: First-Class Timeouts. Our prototype, Node.cure, enables the detection and defeat of all known EHP attacks, with low overhead. Our findings can be directly applied by the EDA community, and we hope they influence the design of existing and future EDA frameworks.

Acknowledgments

We thank the reviewers for their helpful feedback, as well as Adam Doupé for his shepherding. C. Coghlan helped us mine the Snyk.io vulnerability database. J.D. Greef of Rononom suggested the EHP attacks listed in the discussion. A. Kazerooni, S. Rahaman, 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.

References


[22] three.js. https://github.com/webtorrent/three.js


[53] De Groef, W., Devries, D., Nikiforakis, N., and Piessens, F. Flowfox: A web browser with flexible and precise information flow control. CCS.


