The Case of the Poisoned Event Handler: Weaknesses in the Node.js Event-Driven Architecture

James Davis
Virginia Tech
Blacksburg, VA, USA
davisjam@vt.edu

Gregor Kildow
Virginia Tech
Blacksburg, VA, USA
gregor@vt.edu

Dongyoon Lee
Virginia Tech
Blacksburg, VA, USA
dongyoon@vt.edu

ABSTRACT

Node.js has seen rapid adoption in industry and the open-source community. Unfortunately, its event-driven architecture exposes Node.js applications to Event Handler-Poisoning denial of service attacks. Our evaluation of the state of practice in Node.js— combining a study of 353 publicly reported security vulnerabilities and a survey of 151 representative `npm` modules — demonstrates that the community is not equipped to combat this class of attack. We recommend several changes to the state of practice and propose both programming language and runtime approaches to defend against Event Handler-Poisoning attacks.

Keywords

Node.js; Event-driven architecture; Denial of service; ReDoS

1. INTRODUCTION

The Event-Driven Architecture (EDA) is coming into its own. Although EDA approaches have been discussed and implemented in the academic and professional communities for decades (e.g. [17, 7, 18, 13]), historically EDA has only seen wide adoption in user interface settings like web browsers. It is increasingly relevant in systems programming today thanks to the explosion of interest in Node.js[1], an event-driven server-side JavaScript framework.

In applications built using EDA, there is a set of possible *events* (e.g. "mouse click" or "incoming network traffic") for which the developer supplies a set of *Event Handlers* (also known as callbacks); these Event Handlers will be executed when the corresponding events occur [5]. During the execution of such an application, events are generated by external activity and routed to the appropriate Event Handlers.

Figure 1 illustrates the classic EDA formulation, known as AMPED [11]: the operating system or a framework places events in a queue (e.g. through Linux's `poll` system call[2]), and pending events are handled by an *event thread*. The event thread may offload expensive activity to a small, fixed-size *worker pool*, which generates an event to note the completion of each offloaded task. The worker pool is typically implemented using threads or separate processes, and performs services like offering "asynchronous" (blocking but concurrent) access to the file system. This architecture is common to all mainstream general-purpose EDA frameworks today, discussed further in Section 2.2.

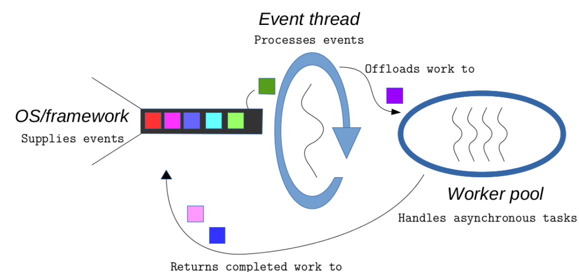

Figure 1: Classic Event-Driven Architecture. Incoming events are handled sequentially by the event thread. The event thread may offload tasks to the worker pool.

Server-side applications implemented using the EDA are vulnerable to a unique type of denial of service attack which we term the *Event Handler-Poisoning* (EHP) attack. Key to the EDA is the use of a small, fixed-size set of Event Handlers that handle each event in turn. In the EDA design of Figure 1, there are two classes of Event Handlers: the event thread, and the workers in the worker pool. If either class of Event Handlers (the event thread, or every worker in the worker pool) becomes blocked, the handling of every pending event will be delayed.

While in a One Thread Per Client (OTPC) setting an expensive request from a client delays only that client, in the EDA context an expensive request delays *every* client. Thus, the EDA model resurrects an old foe: DoS vulnerabilities in an essentially single-threaded server.

Event Handlers might block due to an expensive computation or due to I/O. In our study of the state of practice in the Node.js community in Section 4.3, developers appear to be well apprised of the risks associated with blocking I/O. However, a surprising number of developers appear to take a cavalier approach to the possibility of blocking the event thread

[1] See https://nodejs.org/en/.

[2] Per the Linux man page, "poll...waits for one of a set of file descriptors to become ready to perform I/O."

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

EuroSec'17, April 23 2017, Belgrade, Serbia

© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ISBN 978-1-4503-4935-2/17/04. . . $15.00

DOI: http://dx.doi.org/10.1145/3065913.3065916

with computation, as shown in Section 6.2. The prevalence of EHP vulnerabilities in many common modules suggests the need for reform in the community's practices, and for both programming language and runtime aids to facilitate community adoption.

1.1 Contributions

This paper makes the following contributions:

1. After identifying Node.js as the most popular general EDA framework, we analyze the 353 security vulnerabilities publicized in its package ecosystem, `npm`. We identify and discuss the CPU-bound and I/O-bound EHP vulnerabilities among them.

2. We evaluate the ease with which a Node.js developer can avoid EHP vulnerabilities. Our assessment of development aids and our study of 151 `npm` modules identify significant shortcomings in the tools, documentation, and modules on which a developer might rely.

3. We discuss ways to reduce EHP vulnerabilities in the Node.js ecosystem through changes in the community's practices and by various programming language and runtime defenses.

2. BACKGROUND

In this section we compare the EDA with the OTPC architecture. We then identify the most popular server-side EDA framework for analysis. Lastly, we discuss two types of EHP vulnerabilities in the EDA.

2.1 EDA vs. OTPC

There are two main architectures for scalable web servers. The traditional approach is the OTPC architecture, of which Apache[3] is the most prominent. In the OTPC style, a thread or a process is assigned to each client, with a large maximum number of concurrent clients. The OTPC model isolates clients from one another, reducing the risk of interference. However, each additional client incurs memory and context switching overhead. In contrast, by multiplexing multiple client connections into a single thread and offering a small worker pool for asynchronous tasks, an EDA approach significantly reduces a server's per-client resource use. But the EDA model also exposes the server to instability, e.g. due to vulnerabilities like EHP.

2.2 General-Purpose EDA Frameworks

Though an EDA approach can be implemented in most programming languages, the mainstream general-purpose EDA frameworks for server applications are Java/Reactor[4], C/libuv[5], Python/Twisted[6], Ruby/EventMachine[7], and JavaScript/Node.js[8]. To capture the scope and variety of activity for each framework, we surveyed each of their GitHub[9] repositories using the metrics of number of commits, number of contributors, and number of releases. The results of our GitHub survey are shown in Table 1, and show that Node.js has by far the largest developer community and a significant number of commits.

Framework	Contributors	Commits	Releases
Node.js	1176	15666	399
libuv	247	3643	183
EventMachine	113	1141	27
Reactor	56	4249	42
Twisted	35	20751	38

Table 1: Popularity of event-driven programming environments measured by development activity. Data were obtained from GitHub in December 2016.

Node.js is a server-side event-driven framework for JavaScript that uses the architecture shown in Figure 1. The worker pool is used internally to perform asynchronous DNS queries and file system I/O. In addition, user-defined code can be offloaded to the worker pool. Node.js claims 3.5 million users [1], and its open-source package ecosystem, `npm`, is the largest of any programming language or framework[10].

Though Node.js has seen significant use in industry (e.g. LinkedIn[11], PayPal[12], eBay[13], and IBM[14]), to the best of our knowledge the only academic engagement with security aspects of Node.js applications was Ojamaa et al.'s high-level survey [10]. We believe that more academic involvement in the Node.js community will increase the security of many of the web services we use every day.

2.3 Event Handler-Poisoning Attacks

The key characteristic of the EDA is that many request events are addressed by a small number of Event Handlers. An *Event Handler-Poisoning attack* exploits this design by causing the Event Handlers to block indefinitely, *poisoning* them. This is done through the submission of an expensive task, either CPU-bound or I/O-bound.

We emphasize that in the EDA design of Figure 1, offloading tasks to a fixed-size worker pool is not a panacea for EHP vulnerabilities. Suppose an attacker identifies an EHP vulnerability in code executed by the worker pool. A pool of size k will jam after only k poisonous requests; the EHP vulnerability is essentially the same whether the attack is against the event thread or the worker pool. Therefore, an attacker's aim is to submit requests that cause either the event thread or the entire worker pool to jam up, thereby delaying the handling of subsequent requests.

It should be noted that an attack that jams the worker pool can in principle be carried out against any architecture or system with a cap on the amount of resources available for clients, including both EDA and pool-based OTPC designs. However, the connection pool in a OTPC system is significantly larger than the worker pool in typical EDA implementations. While Node.js has a default worker pool of

[3]See http://www.apache.org/.
[4]See http://projectreactor.io/.
[5]See http://libuv.org/.
[6]See http://twistedmatrix.com/.
[7]See http://rubyeventmachine.com/.
[8]See http://nodejs.org/.
[9]See https://github.com/.
[10]See http://www.modulecounts.com/.
[11]See http://venturebeat.com/2011/08/16/linkedin-node/.
[12]See https://www.paypal-engineering.com/2013/11/22/node-js-at-paypal/.
[13]See http://www.ebaytechblog.com/2013/05/17/how-we-built-ebays-first-node-js-application/.
[14]See https://developer.ibm.com/open/openprojects/node-red/.

size 4 with a maximum of 128 workers[15], Apache's `httpd` server has a default connection pool of size 256-400, with a maximum size of around 20000[16]. Due to the size of an OTPC server's connection pool, attempts to jam it would presumably be detected by an IDS or addressed through rate limiting. By comparison, the small number of requests needed to jam a vulnerable EDA server's worker pool is too few to draw attention from network-level defenses, placing the burden of defense on the application developer or the runtime system to address it.

An important facet of an EHP attack is its asymmetry. As in a DDoS attack, a carefully crafted "evil request" will be roughly the same size and shape as the requests submitted by legitimate clients. However, unlike a DDoS attack, the attacker need only submit one (or k) rather than a flood. Cambiaso et al. capture this concept with the term "slow DoS attacks" [3].

2.3.1 CPU-bound Vulnerabilities

The CPU-bound EHP vulnerabilities in an EDA system are a form of Algorithmic Complexity (AC) vulnerability [4]. In an AC attack, attackers force victims to take longer than expected to perform an operation, typically by exploiting the Worst-Case Execution Time (WCET) of an algorithm used to process requests. Rather than overwhelming the victim with volume, the attacker uses the victim's vulnerable algorithms against him. If an attacker identifies an AC-style vulnerability in an EDA-based server, he can submit "evil input" to trigger the WCET of the vulnerable algorithm. When an Event Handler picks up the request, it will take an inordinately long time to handle it, starving subsequent requests – an EHP attack.

We argue that CPU-bound EHP vulnerabilities in the EDA are more general than Crosby and Wallach's formulation of AC attacks. Per [4], the victim of an AC attack is an algorithm or data structure that performs well in the average case but poorly on carefully chosen "evil inputs". In the EDA setting, however, *any* algorithm with *non-constant WCET* (non-C-WCET) and *unbounded input* constitutes an EHP vulnerability.

While the EDA was originally intended for I/O-bound applications, we argue that the need to perform computation in an EDA system is growing. Full-featured servers that rely on the EDA must inevitably deal with the computations needed to implement business logic, and with the advent of fog computing [2], embedded IoT devices will increasingly be called on to perform data processing. As these input-driven devices are most readily programmed using the EDA model, as shown in Cylon.js[17], processing the incoming data will require computation within an EDA setting.

2.3.2 I/O-bound Vulnerabilities

Web servers are commonly called on to interact with the file system. An I/O-bound EHP vulnerability exists when the attacker can request the server make a particularly slow I/O. For example, in a file server, a slow I/O might be a read of an unusual file, which can take on average 8x longer to serve [20], or perhaps data stored on slow media, e.g. a network drive. We suggest several ways in which an attacker might exploit an I/O-bound EHP in Section 4.3.

[15]See http://docs.libuv.org/en/v1.x/threadpool.html.

[16]See values for ServerLimit and MaxRequest Workers in the Apache `httpd` documentation, available at https://httpd.apache.org/docs/2.4/mod/

[17]See https://cylonjs.com/.

3. C-WCET-PARTITIONED ALGORITHMS

When an algorithm's overall complexity is not $O(1)$, being C-WCET-partitioned gives it two properties that make it ideal for the EDA setting. First, it will be *cooperative*; it will not be executed atomically, but will instead allow other events to have a turn on the Event Handler. Second, it will be *predictable*: other events will have a turn on the Event Handler within a fixed amount of time, necessary for high throughput in a context like Node.js (where events of different types and costs are handled by a single worker pool). A C-WCET-partitioned algorithm can block neither the event thread nor the worker pool, and thus an application composed of C-WCET-partitioned algorithms is invulnerable to EHP attacks. To the best of our knowledge we are the first to propose this hardline stance on C-WCET partitioning for EDA applications.

C-WCET partitioning is essentially the logical extreme of cooperative multitasking [15], and interprets EDA systems as real-time (embedded) systems with a need for strict forward progress guarantees [19]. C-WCET partitioning also makes predictable the worst-case overall completion time of a request, a useful feature for application designers. In Sections 4 and 6 we discuss the security vulnerabilities inherent in the use of non-C-WCET-partitioned algorithms in the EDA setting.

4. KNOWN EHP VULNERABILITIES IN NPM

Having described the concept of an EHP attack against EDA-based applications, we now assess the documented vulnerabilities of this type in the Node.js ecosystem. The open-source `npm` community gathers the bulk of its vulnerability data via self-reporting, aggregated by the Node Security Platform (NSP)[18] and Snyk.io[19] in public databases. In this section we provide an analysis of the vulnerabilities reported by these two agencies, focusing on EHP vulnerabilities.

4.1 Known Vulnerability Classification

Initial analysis showed that Snyk.io's database was a superset of NSP's, so we examined all vulnerabilities in the Snyk.io database as of February 1, 2017. Snyk.io has 353 such vulnerabilities, of which 161 had not been assigned a CWE label. We therefore created an appropriate set of high-level categories, placing each vulnerability into one of these categories and obtaining the distribution shown in Figure 2.

We draw the reader's attention to three elements of the distribution in Figure 2. First, the 20 *DoS by AC* vulnerabilities are due to Node.js's EDA design. In a OTPC environment, such vulnerabilities would be annoying but not catastrophic; in Node.js's EDA design, they represent a CPU-bound EHP vulnerability leading to DoS. Second, the 14 *Directory traversal* and 48 *Code injection* vulnerabilities are not specific to the EDA, but in the EDA setting some of these vulnerabilities comprise CPU-bound and I/O-bound EHP vulnerabilities. Lastly, since `npm` has approximately 400,000 modules[20], we were surprised that only 353 vulner-

[18]See https://nodesecurity.io/.

[19]See https://snyk.io/.

[20]See https://www.npmjs.com/.

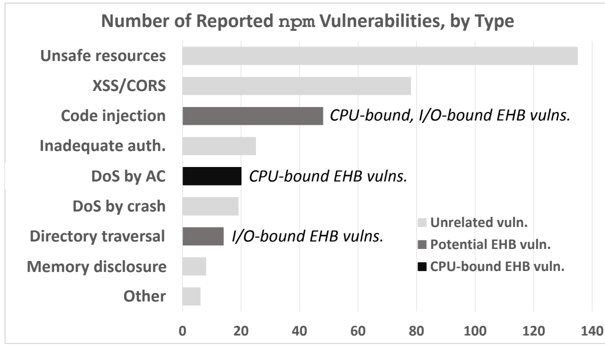

Figure 2: Classification of the known 353 `npm` module vulnerabilities. The "Unsafe resource" vulnerabilities are trivial cases in which the module obtains resources over HTTP (not HTTPS), allowing a man-in-the-middle attack.

abilities have been reported. Caution dictates that many more security vulnerabilities remain hidden. We are therefore uncertain of the total number and relative distribution of security vulnerabilities in `npm` modules, and believe this field is ripe for further study.

4.2 CPU-bound EHP Vulnerabilities

We will first discuss the *DoS by AC* vulnerabilities, and then explore the EHP possibilities of the *Code injection* vulnerabilities.

General algorithmic complexity vulnerabilities In one of these vulnerabilities, the report observed that a module's APIs had $O(n)$ WCET, and that for large input this API would block an Event Handler.

ReDoS vulnerabilities The other 19 such vulnerabilities were Regular expression Denial of Service (ReDoS) vulnerabilities [14]. Regular expression engines are frequently implemented with worst-case exponential-time performance[21] to support advanced features like back references[22]. A *vulnerable* regular expression triggers the exponential-time worst-case performance of the engine on evil input. We discuss these vulnerabilities further in Section 5.

***DoS by AC*: fix approaches** The patch for the $O(n)$ algorithm was to truncate the input to a maximum size, achieving $O(1)$ WCET at the cost of generality. For the remaining 19 ReDoS vulnerabilities, resolution took three main forms: replacing the vulnerable regular expression with a safe one (9 cases), filtering the input by type or size prior to feeding it to the regular expression (4 cases), and removing the offending regular expression(s) (3 cases). One fix employed a combination of solutions; in the last four, the vulnerability remains unrepaired.

Code injection vulnerabilities These 48 vulnerabilities expose Node.js servers to an array of exploit possibilities, including both CPU-bound and I/O-bound EDA attacks. For example, the code `while(1);` [16] constitutes a CPU-bound EDA attack.

4.3 I/O-bound EHP Vulnerabilities

14 of the vulnerabilities were *Directory traversal*, in which the attacker can try to read from and possibly write to any files in the file system. We propose two ways to exploit these vulnerabilities to carry out an EHP attack on Linux. We call these exploits *Eternal I/O* attacks, and to the best of our knowledge these exploits are novel.

While several CVE[23] reports suggest the possibility of DoS caused by an infinite read from `/dev/zero`, such a request will not constitute an EHP attack against properly written Node.js code. Node.js's standard file system `Stream` APIs cooperatively partition I/O requests into a sequence of fixed-size I/Os, dividing a read from `/dev/zero` into an infinite number of fast partitions. As such, a `/dev/zero` attack will delay but not block the victim Event Handler, though ENOMEM conditions may result.

Unfortunately, Node.js's file system `Stream` implementation is not *C-WCET* partitioned. The underlying file descriptor is blocking, so I/O against a slow, blocking source will occupy the Event Handler for long intervals in each partition (each `read()`/`write()`). For example, reading from `/dev/random` will poison the Event Handler[24], as will reading from an empty FIFO or writing to a full one.

5. IDENTIFYING VULNERABLE REGULAR EXPRESSIONS

Given the clear potential for ReDoS vulnerabilities, we wanted to understand how readily a Node.js developer can determine whether his regular expressions are vulnerable. To this end, we extracted the known-vulnerable regular expressions from the Snyk.io vulnerability reports and analyzed them using the tools available to the community.

Extracting these regular expressions was non-trivial. The ReDoS vulnerability reports from NSP and Snyk.io do not indicate the vulnerable regular expression(s) or the evil inputs, even after a patch is available. Consequently, we identified vulnerable or potentially-vulnerable regular expressions through manual examination of the vulnerability patches and the module source code. As we could not always pinpoint the specific vulnerable regular expression, we extracted 44 *potentially* vulnerable regular expressions.

We tested each regular expression using the only open-source regular expression analysis tools we could identify: `safe-regex`[25] and `rxxr2` [12]. `safe-regex` labels as vulnerable all regular expressions with star height [8] greater than one, while `rxxr2` performs a more complex analysis based on an idealized backtracking regular expression engine. `rxxr2` did not support unicode characters and unescaped meta characters, and we transformed any regular expressions using these features into semantically equivalent expressions that it could parse. `rxxr2` reports evil input it believes will trigger exponential-time performance for the regular expression, simplifying validation of its report, while `safe-regex` does not.

Disagreement between oracles These static analysis tools disagreed on many of the 44 regular expressions we extracted using the vulnerability reports. Figure 3 shows the extent to which the oracles agreed in their assessments of "vulnerable".

Both oracles were inaccurate in the Node.js environment. `rxxr2` was complete (no false positives) but not sound (it had

[21]See http://lh3lh3.users.sourceforge.net/reb.shtml.

[22]A back reference n matches the substring in the nth parenthetical in the regular expression.

[23]See http://cve.mitre.org/.

[24]See `man(4) random`.

[25]See https://github.com/substack/safe-regex/.

Figure 3: The `safe-regex` and `rxxr2` oracles disagreed on whether many of the regular expressions were vulnerable. Each oracle found true vulnerabilities the other did not.

false negatives). `safe-regex` was not sound, and we do not know whether it was complete. We confirmed that each of the 15 regular expressions identified as unsafe by `rxxr2` was vulnerable to ReDoS. In addition, we manually crafted evil input based on the analysis of `safe-regex`, and were successful in demonstrating two of these regular expressions to be vulnerable; we believe a few to be safe (false positives) and are not yet certain on the rest. We are working on automatically producing evil input based on `safe-regex`'s analysis in order to automatically evaluate the accuracy of its analysis.

We expected inaccuracies in `safe-regex`, as the tool's documentation explains that it is best-effort only. The unsoundness of `rxxr2` [12] was more surprising, suggesting either bugs in the authors' implementation or assumptions about an idealized regular expression engine that do not hold for Node.js.

6. CPU-BOUND EHP VULNERABILITIES IN THE WILD

Based on our assessment of the prevalence of EHP vulnerabilities in Section 4, we performed a preliminary study of `npm` vulnerabilities "in the wild."

6.1 ReDoS Vulnerabilities

Keeping in mind the common use of regular expressions to parse input, we wanted to know whether the regular expressions in the most popular `npm` modules were vulnerable. We examined the first two pages of "most starred" packages on `npm`, yielding 71 modules with GitHub repositories. We consider these modules to be well-maintained and likely to be compatible with a recent LTS version of Node.js, ensuring consistency in analysis.

We instrumented a version of Node.js v6.9.4 (a LTS version of Node.js) to emit regular expressions as they were declared. We ran the test suite of each of the modules using our version of Node.js, extracting 16,252 unique regular expressions from a total of 487,063 declarations.

We applied the oracles discussed in Section 5 to determine whether these regular expressions are vulnerable. Preliminary results produced over 700 potential vulnerabilities from `safe-regex` and 20 from `rxxr2`, of which we have identified several as truly vulnerable to ReDoS.

6.2 Non-C-WCET Algorithms

As argued in Section 3, in the Node.js EDA, only C-WCET-partitioned algorithms are safe from EHP vulnerabilities. We examined 80 `npm` modules in detail to determine how they handle potentially computationally expensive tasks, manually inspecting both their documentation and their implementations. We chose the first 20 modules returned by `npm` searches for `string`, `string manipulation`, `math`, and `algorithm`, in hopes of striking a balance between relevance (high usage) and potential algorithmic complexity.

Our initial findings were troubling. Only two of the hundreds of APIs we evaluated were partitioned, and one of these synchronously executed an exponential-time algorithm before yielding control. The majority of the API documentation did not state the running time of the synchronous algorithms, and by inspection we identified algorithms with complexities ranging from $O(n)$ to $O(2^n)$. The modules we selected were not pet projects; combined, these 80 modules were downloaded over 200 million times in December 2016 alone. It seems clear to us that the majority of module developers are not cognizant of the risks of unbounded non-C-WCET-partitioned algorithms, and that many Node.js applications are therefore likely vulnerable to CPU-bound EHP attacks. These findings are in line with previous work studying the frequency of asynchronous callbacks in server-side JavaScript, though we identified a lower proportion of asynchronous callbacks than they did [6].

7. DISCUSSION AND RESEARCH DIRECTIONS

In this section we share suggestions for the Node.js development community, and we propose five research directions to combat EHP attacks in the EDA setting.

7.1 Community Recommendations

Vulnerability reporting Many reports from Snyk.io and NSP lacked details like CWE labels or CVE IDs, a precise description of the vulnerability, a proof of concept, patch information, or the version(s) of Node.js on which the vulnerability exists. Including this information in reports would ease subsequent analyses of these rich sources of vulnerabilities.

`npm` module documentation Given the security implications of poisoning an Event Handler, we strongly recommend that `npm` modules clearly document their WCET. While this information would be useful for developers in any setting, it is vital for developers building secure EDA-based systems.

Use multiple regex oracles We found both `rxxr2` and `safe-regex` to be inaccurate. Their combined wisdom, however, appeared effective at identifying vulnerable regular expressions, and we recommend developers query both of these oracles to ensure an accurate report. We will be releasing tools to facilitate these queries and to validate the oracles' responses.

7.2 Potential Research Directions

C-WCET-partitioned algorithms As argued in Section 3, algorithms used in an EDA system like Node.js should be C-WCET partitioned; this concept underlies all of the EHP vulnerabilities discussed in Sections 4 and 6. Happily, C-WCET partitioning can generally be realized by repeatedly calling a kernel function, a straightforward technique in JavaScript thanks to its notion of closures. Would an extensive set of `npm` modules with C-WCET-partitioned algorithms show the Node.js community the way forward? Could a speculative execution scheme safely alleviate the need for C-WCET partitioning in part or in whole? Can we automatically refactor algorithms to be C-WCET partitioned? Lin et al. take a step towards this refactoring in Android [9],

though their technique is inappropriate for Node.js because of Node.js's fixed-size worker pool.

TimeoutExceptions Alongside C-WCET partitioning, we recommend the introduction of runtime-generated `TimeoutException`s into EDA implementations. A `TimeoutException` would abort synchronous computation after a specified time limit, allowing developers to use open-source modules while still ensuring protection against EHP vulnerabilities.

A hybrid regex engine Except for back references, every feature of JavaScript regular expressions can be supported by a linear-time regular expression engine. As back references are little-used in practice (of the 16,252 we analyzed in Section 6.1, only 54 used them), we recommend that Node.js offer a linear-time engine for the common case. This technique has also been discussed by Cox[26].

To demonstrate the potential of the hybrid approach, we evaluated each of the `rxxr2`-flagged regular expressions from Section 4.2 against Node.js and against the linear-time `RE2`[27] engine. While Node.js required exponential time to process evil input, `RE2` performed in linear time.

An accurate oracle As discussed in Section 4.2, neither of the regular expression oracles we used was accurate, presumably due to incorrect assumptions about the implementation of the Node.js regular expression engine. Can we develop a regular expression oracle tailored to individual regular expression engine implementations, perhaps based on their feature sets and a few implementation details?

Truly asynchronous file I/O Node.js implements asynchronous file I/O by offloading synchronous file I/O operations to its worker pool. Each time the worker pool is assigned an Eternal I/O (Section 4.3), an Event Handler is poisoned. We are unaware of any existing solution to this problem. Were this I/O implemented using true kernel asynchronous IO or non-blocking file descriptors, Eternal I/Os that make no forward progress would consume operating system resources but would *not* poison an Event Handler.

8. CONCLUSION

The Event-Driven Architecture holds significant promise for scaling, and Node.js developers can get started quickly and re-use an enormous amount of existing code written for client-side JavaScript. However, EDA-based servers are vulnerable to Event Handler-Poisoning DoS attacks, just like old single-threaded servers were. As demonstrated by our analysis of known vulnerabilities and our survey of potential new ones, the Node.js community seems to be incautious about this vulnerability. We have proposed a variety of possible defenses against EHP attacks, and look forward to assessing their success under various threat models.

Acknowledgments

The authors thank Long Cheng and Dr. Danfeng Yao for their suggestions, and Talha Ghaffar and M. Usman Nadeem for their help on our `npm` hunting expedition. We are also grateful to the reviewers for their feedback.

9. REFERENCES

[1] New Node.js Foundation Survey Reports New "Full Stack" In Demand Among Enterprise Developers, 2016. https://nodejs.org/en/blog/announcements/nodejs-foundation-survey/.

[2] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. Fog Computing and Its Role in the Internet of Things. In *Mobile Cloud Computing (MCC)*, 2012.

[3] E. Cambiaso, G. Papaleo, and M. Aiello. Taxonomy of Slow DoS Attacks to Web Applications. *Recent Trends in Computer Networks and Distributed Systems Security*, pages 195–204, 2012.

[4] S. A. Crosby and D. S. Wallach. Denial of Service via Algorithmic Complexity Attacks. In *USENIX Security*, 2003.

[5] S. Ferg. Event-driven programming: introduction, tutorial, history. 2006.

[6] K. Gallaba, A. Mesbah, and I. Beschastnikh. Don't Call Us, We'll Call You: Characterizing Callbacks in Javascript. In *International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2015.

[7] D. Goodman and P. Ferguson. *Dynamic HTML: The Definitive Reference*. O'Reilly, 1 edition, 1998.

[8] K. Hashiguchi. Algorithms for determining relative star height and star height. *Information and Computation*, 78(2):124–169, 1988.

[9] Y. Lin, C. Radoi, and D. Dig. Retrofitting Concurrency for Android Applications through Refactoring. In *ACM International Symposium on Foundations of Software Engineering (FSE)*, 2014.

[10] A. Ojamaa and K. Duuna. Assessing the security of Node.js platform. In *7th International Conference for Internet Technology and Secured Transactions (ICITST)*, pages 348–355, 2012.

[11] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An Efficient and Portable Web Server. In *USENIX Annual Technical Conference (ATC)*, 1999.

[12] A. Rathnayake and H. Thielecke. Static Analysis for Regular Expression Exponential Runtime via Substructural Logics. *CoRR*, 2014.

[13] R. Rogers, J. Lombardo, Z. Mednieks, and B. Meike. *Android application development: Programming with the Google SDK*. O'Reilly Media, Inc., 1 edition, 2009.

[14] A. Roichman and A. Weidman. VAC - ReDoS Regular Expression Denial Of Service. *OWASP*, 2009.

[15] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. Wiley Publishing, 9th edition, 2012.

[16] B. Sullivan. Server-Side JavaScript Injection. *BlackHat USA*, (July):1–7, 2011.

[17] R. E. Sweet. The Mesa programming environment. *ACM SIGPLAN Notices*, 20(7):216–229, 1985.

[18] M. Welsh, D. Culler, and E. Brewer. SEDA : An Architecture for Well-Conditioned, Scalable Internet Services. In *SOSP*, 2001.

[19] R. Wilhelm, J. Engblom, A. Ermedahl, et al. The Worst-Case Execution-Time Problem - Overview of Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):36, 2008.

[20] Z. Wu, M. Xie, and H. Wang. Energy Attack on Server Systems. In *USENIX WOOT*, 2011.

[26] See https://swtch.com/ rsc/regexp/regexp1.html.

[27] See https://github.com/google/re2.