Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers

Cristian-Alexandru Staicu and Michael Pradel

Technical Report TUD-CS-2017-0305

TU Darmstadt, Department of Computer Science

December, 2017

TECHNISCHE UNIVERSITÄT DARMSTADT

SOLA SoftwareLab

Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers

***Abstract*—Regular expression denial of service (ReDoS) is a class of algorithmic complexity attacks where matching a regular expression against an attacker-provided input takes unexpectedly long. The single-threaded execution model of JavaScript makes JavaScript-based web servers particularly susceptible to ReDoS attacks. Despite this risk and the increasing popularity of the server-side Node.js platform, there is currently little reported knowledge about the severity of the ReDoS problem in practice. This paper presents a large-scale study of ReDoS vulnerabilities in real-world web sites. Underlying our study is a novel methodology for analyzing the exploitability of deployed servers. The basic idea is to search for previously unknown vulnerabilities in popular libraries, hypothesize how these libraries may be used by servers, and to then craft targeted exploits. In the course of the study, we identify 25 previously unknown vulnerabilities in popular modules and test 2,846 of the most popular websites against them. We find that 339 web sites (11% of the ones that use Express, a popular server-side JavaScript framework) suffer from at least one ReDoS vulnerability and some even suffer from multiple ones. A single request can block a vulnerable site for several seconds, and sometimes even much longer, enabling denial of service attacks that pose a serious threat to the availability of these sites. We also show that the fact whether a website is vulnerable is independent of its popularity, indicating that the problem requires attention across a wide spectrum of web providers. Our results are a call-to-arms for developing techniques to detect and mitigate ReDoS vulnerabilities in JavaScript.**

I. Introduction

Regular expressions are widely used in all kinds of software. Since regular expressions are easy to get wrong [38], which may help attackers to bypass checks [15], [5], developers are trained to think about the correctness of regular expressions. In contrast, another security-related aspect of regular expressions is often neglected: the performance, specifically, how long it takes to match a string against the regular expression. Unfortunately, given a specifically crafted input, matching against a suboptimally designed regular expression can easily take several minutes or even hours. For example, matching the apparently harmless regular expression `/(a+)+b/` against a sequence of 30 "a" characters on the Node.js JavaScript platform takes about 15 seconds on a standard compute.[1] Matching a sequence of 35 "a" characters already takes over 8 minutes, i.e., the matching time explodes exponentially.

If a server implementation suffers from this kind of performance problem, then an attacker can exploit it to overwhelm the server with hard-to-match inputs. This attack is known as *regular expression denial of service*, or short *ReDoS*. Such attacks are a form of algorithmic complexity attack [10] that exploits the worst-case complexity behavior of algorithms that match a string against a regular expression. Since for some regular expressions, the worst-case complexity is much higher than the average-case complexity, an attacker can cause denial of service with a few, relatively small inputs. For example, given a web server that suffers from ReDoS, a single request with only a few bytes may block a worker thread for a long time, providing a low-bandwidth attack vector.

Even though ReDoS has been known for several years, recent developments in the web server landscape bring new and increased attention to the problem. The reason is that JavaScript is becoming increasingly popular not only for the client-side but also for the server-side of web applications. However, the single-threaded nature of JavaScript makes server applications much more susceptible to ReDoS attacks. In a JavaScript-based web server, e.g., based on the popular Node.js and Express platforms, every request is handled by a single thread of execution. In practice, to avoid making the server unresponsive by blocking this thread, developers try to split any long-running computation into smaller events, which are than handled asynchronously. The problem is that in current JavaScript engines, matching a string against a regular expression cannot be easily split into multiple chunks of computation. As a result, a single request can effectively block the main thread, making the web server unresponsive to any other incoming requests and preventing it from finishing any other already established requests.

Despite the importance of ReDoS in web servers, there is currently little reported knowledge about the prevalence of ReDoS vulnerabilities in real-world websites. In this paper, we present the first comprehensive study of ReDoS across a large number of websites. Our study involves 2,846 of the most popular websites. We seek to answer the following questions:

- How widespread are ReDoS vulnerabilities in the server-side part of real-world JavaScript-based websites?
- What is the effect of vulnerabilities on the response time of web servers?
- What kinds of vulnerabilities are the most prevalent?
- Are more popular websites less vulnerable to ReDoS?
- Are existing defense mechanisms in use and if so, how effective are they in preventing ReDoS attacks?

Answering these questions involves solving two non-trivial methodological challenges. First, how to identify ReDoS vulnerabilities in the server-side of websites when their source code is not available. We address this challenge by identifying a set of previously unknown vulnerabilities in popular libraries

[1]We use JavaScript syntax for regular expressions, i.e., a pattern is either enclosed by slashes or given to the `RegExp()` constructor.

and by speculating how these libraries may be used in servers. In this process, we apply a combination of automated and manual analysis to 10,000 software packages and found 25 vulnerabilities and eight exploits. Second, how to analyze which websites are exploitable without actually performing a denial of service attack against live websites. We address this challenge by triggering requests with increasing input size, using both manually crafted exploit inputs and randomly generated, harmless inputs, and by statistically comparing the response times.

Using this methodology, we identify 339 websites that suffer from at least one ReDoS vulnerability. Based on experiments with locally installed versions of the vulnerable server-side libraries, attacking these websites with crafted inputs can cause a web server to remain unresponsive for several seconds or even minutes. These problems are due to a very small number of vulnerabilities, with a single vulnerability that causes 241 sites to be exploitable. While this is encouraging from a mitigation point of view, it also implies that an attacker aware of a single, previously unknown vulnerability can cause serious harm to vulnerable websites.

Some of the vulnerabilities we identify are more serious that the others. For one of them, 50 characters of carefully created input can block the main thread for 10 minutes, while for most of the others at least 10,000 characters are needed to trigger one second of slowdown. Since we use HTTP header values to transport payloads, limiting the size of the header can serve as a first line of defense. We find that many websites implement this defense mechanism: 85% of the websites reject headers longer than 25,000 characters and 3% even reject headers longer than 500 characters. However, limiting the header size alone is insufficient to defend against ReDoS because even millisecond-level matching times can be used to severely degrade the performance of a website, and because there are other ways to transport payloads.

Ojamaa and Düüna [24] were the first to identify ReDoS as a threat for the Node.js platform. Davis et al. [11] confirm that such problems exist in popular modules and report that 5% of the security vulnerabilities identified in Node.js libraries are ReDoS. No prior work has studied the impact of ReDoS on real-world web sites. Existing work on detecting ReDoS vulnerabilities mostly targets languages other than JavaScript. For example, Wüstholz et al. [39] propose a static analysis of ReDoS vulnerabilities in Java. The only available tool for JavaScript that we are aware of is a small utility called `safe-regex`[2], which checks for simple AST-level patterns known to cause ReDoS. However, this approach is notoriously prone to both false positives and false negatives, since it reasons neither about the context in which these patterns appear nor about the actual performance of regular expression matching. Our work shows the urgent need for effective tools and techniques that detect and prevent ReDoS vulnerabilities in JavaScript.

In summary, this paper contributes the following:

[2]https://www.npmjs.com/package/safe-regex

- A novel methodology for analyzing the exploitability of deployed servers. The key ideas are (i) to hypothesize how server implementations may use libraries that have previously unknown vulnerabilities and (ii) to assess whether an attack is feasible without actually attacking the servers.
- The first comprehensive study of ReDoS vulnerabilities in JavaScript-based web servers. Out of thousands of studied websites, we find over 10% to be vulnerable, including many popular sites that are vulnerable to a very serious form of ReDoS.
- Empirical evidence that ReDoS is a real and widespread threat. Our work calls for novel tools and techniques that detect and prevent ReDoS vulnerabilities.
- A benchmark of previously unreported ReDoS vulnerabilities and ready-to-use exploits, which we make available for future research on finding, fixing, and mitigating ReDoS vulnerabilities:

https://github.com/sola-da/ReDoS-vulnerabilities

II. Background

A. Regular Expression Matching

Regular expressions are a popular way to specify patterns of characters. The main operation is to check whether a given sequence of characters *matches* a specified pattern. There are multiple ways to approach this problem; we here focus on the most popular one, which is used in most modern programming languages. Given a regular expression, the first step is to convert it into a finite automaton. The automaton consists of a starting state, one or more accepting states, set of intermediate states, and a set of transitions. Each transition can either consume one character from the input or not (ϵ transition). Given an automaton, the problem of matching a regular expression against a string is equivalent to finding a sequence of transitions from the initial state to an accepting state that consumes all the characters in the string.

For example, consider the regular expression `/^(a+b)?$/` and its equivalent automaton in Figure 1. One way to obtain such an automaton is Thompson's construction [34]. Given the string "aab", the automaton starts from state s and has two available transitions, to states 1 and 3. It first takes the transition to state 1 from where the only available transition is into state 2. From there on, it transitions to the accepting state a. Since the input string was not consumed and there are no available transitions, the algorithm backtracks to a state with not yet explored alternatives, in our case, to state s. From state s, it will transition to state 3 than 4 and so on. After multiple explorations the algorithm identifies the sequence of transitions $s \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow a$, which reaches the accepting state and consumes all characters of the input string.

B. Regular Expression Denial of Service (ReDoS)

The backtracking style algorithm is not the most efficient approach to regular expression matching, but it is the most

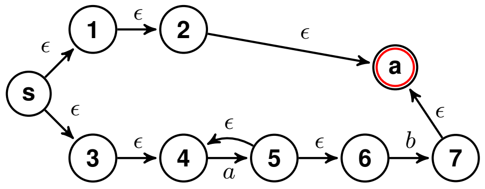


Fig. 1: Automaton for the regular expression `/^(a+b)?$/`. s is the starting state and a is the accepting state.

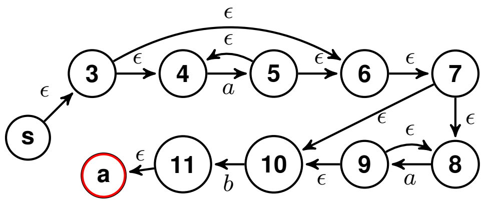


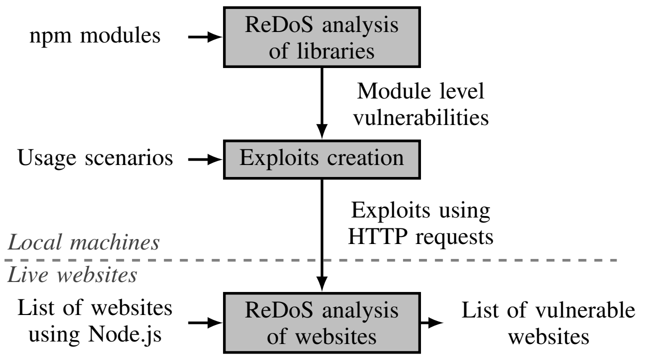
Fig. 2: Automaton for the regular expression `/^a*a*b$/`. s is the starting state and a is the accepting state.

used one due its ease of implementation and the expressive power it offers. The downside is that for some regular expressions and inputs, the algorithm needs to backtrack a possibly large number of times. ReDoS attacks exploit these pathological cases.

For example, consider the regular expression `/^a*a*b$/`, its automaton in Figure 2, and the input string "aaa". Each character "a" can be matched using two transitions, $4 \rightarrow 5$ and $8 \rightarrow 9$. At each step, the algorithm needs to decide which of these two transitions to take. Eventually, since there is no character "b" in the input string, the algorithm will always fail when reaching state 11. However, before concluding that the input string does not match the pattern, the algorithm tries all possible ways of matching the "a" characters, i.e., it backtracks three times. The example is a regular expression of super-linear complexity [39], since the number of transitions during matching is quadratic in the input size. Other regular expression even have exponential complexity, e.g., because of nested repetitions, such as in `/^(a*)*b$/`. In our study, we identify ReDoS vulnerabilities of both these types and show that both are of importance for server-side JavaScript.

C. Server-side JavaScript

JavaScript has become one of the most popular programming languages, but it was traditionally used for client-side tasks only. Recently, the idea of using JavaScript on the server-side gained traction with the wide adoption of Node.js, a single-threaded, event-based platform that uses asynchronous I/O calls. In Node.js, the main thread of execution runs an event loop, called the *main loop* that handles events as they are dispatched by network requests, I/O operations, timers, etc. The overall performance of the system is highly dependent on the amount of computation done in the main thread. Specifically, a slow computation triggered in the main loop by a request, slows down all the other incoming requests. For example, matching a string against a regular expression


npm modules
ReDoS analysis of libraries
Module level vulnerabilities
Usage scenarios
Exploits creation
Exploits using HTTP requests
Local machines
Live websites
List of websites using Node.js
ReDoS analysis of websites
List of vulnerable websites
Fig. 3: Overview of the methodology.

with quadratic or exponential complexity slows down all other requests.

Since in JavaScript-based servers, regular expressions are matched in the main loop, a ReDoS-based attack can be much more harmful than in multi-threaded web servers, such as Apache. For example, consider a regular expression that takes more than an hour to match. As we will show in the evaluation, such expressions exist in widely used JavaScript software. To completely block an Apache web server, we need to send hundreds of such requests, each blocking one thread. Depending on the number of available parallel processing units, the operating system, and the thread pool size, new requests can still be handled even with hundred of busy threads running. In contrast, in Node.js one such request is enough to completely block the server for an hour. To make matters worse, even less severe ReDoS payloads can significantly degrade the availability of a Node.js server, as we show in Section IV-C.

III. Methodology

This section presents our methodology for studying ReDoS vulnerabilities in real websites. The overall goals of the methodology are to understand (i) how widespread such vulnerabilities are, (ii) whether an attacker could exploit them to affect the availability of live websites, and (iii) to what extent existing defense mechanisms address the problem. To answer these questions, our methodology must address two major challenges. The first challenge is a technical problem: Since the server-side source code of most websites is not available, how to know what vulnerabilities a website may suffer from? The second challenge is an ethical concern: How to study the potential impact of attacks on live websites without actually causing noticeable harm to these websites?

Figure 3 shows a high-level overview of the methodology. We address the two challenges through experiments performed on machines under our control and on live websites. A main insight to address the first challenge is to search for vulnerabilities in popular JavaScript libraries and to speculate how servers may use these libraries. More precisely, we analyze third-party libraries, called node package manager modules (npm packages or npm modules for short), to find vulnerabilities that may be exploitable via HTTP requests. We then hypothesize how the server implementation may use these packages and create exploits for these scenarios.

To address the second challenge, we present a technique that tests whether a site is vulnerable but that avoids blocking the site for a noticeable amount of time. The basic idea is to start with very small payloads that do not require more computation time than normal web requests, and to then slowly increase the payload – just long enough to claim with confidence that the site *could* be exploited if an attacker used larger payloads. To decide on the size of payloads sent to live websites, we run initial experiments on locally installed web servers that use the vulnerable packages.

A. *Identifying Websites with Server-side JavaScript*

We consider the most popular one million websites aggregated by Alexa[3] as candidate targets for our study. Many of these websites do not use JavaScript on the server-side and analyzing all the websites against our exploits is prohibitive. Instead, we select sites that run a JavaScript-based web server, more precisely the popular *Express* framework.[4] Express is the most popular server-side JavaScript framework and our methodology can be easily applied to other frameworks.

To identify websites running Express, we make a request to each of the one million websites and check whether the header `X-Powered-By` is "Express". The framework sets this value by default on a fresh installation. In total, 2,846 sites set this header which account for a market share of around 0.3%, consistent with current estimates by others.[5] In Figure 4, we show the number of Express-based websites in a batch of 100,000 sites, ordered by popularity. We observe that Express tends to be used by the more popular websites, confirming the importance of studying the security of JavaScript-based servers.

The scope of our study is likely to underapproximate the set of JavaScript-based sites due to multiple reasons. First, headers may be filtered to prevent attackers from performing targeted attacks, such as ours. Second, there are frameworks other than Express for server-side JavaScript. Given this underapproximation, our work provides a lower bound for the impact that ReDoS vulnerabilities have in practice.

B. *Finding ReDoS Vulnerabilities in Libraries*

Similar to previous work [39], we consider a regular expression to be vulnerable if we can construct inputs of linearly increasing size that cause the matching time of the expression to increase super-linearly. One may argue that such a loose definition for a vulnerability produces a lot of false positives that are harmless from a security point of view. As we will show in our evaluation, this is not the case, because even an only quadratic input dependency can be used to block the main loop for multiple seconds.

[3]http://www.alexa.com/
[4]https://expressjs.com/
[5]https://w3techs.com/technologies/details/ws-nodejs/all/all

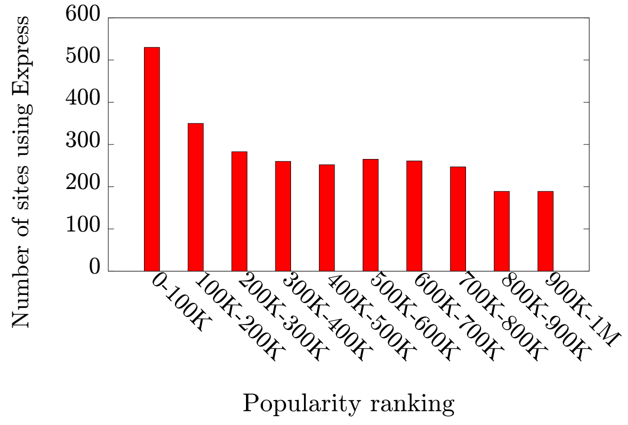


Fig. 4: Number of server-side JavaScript websites within a given popularity range. Each bar shows the number of websites using the Express framework in a range of 100,000 websites.

Our methodology relies on knowing previously unknown, or at least not yet fixed, ReDoS vulnerabilities in popular npm modules. We use a combination of automated and manual analysis to identify exploitable ReDoS vulnerabilities, similar to what a potential attacker might do. At first, we download the most popular modules and extract their regular expressions using a traversal of the abstract syntax trees (ASTs) of the JavaScript code. Next, we perform different queries on the database of extracted expressions to find specific patterns that are known to be vulnerable. For example, we search for expressions containing repetitions of a negated group followed by a character. An output of this query is the second regular expression in Figure 8, which contains the subexpression `[^=]=`. A regular expression that is not anchored with a start anchor and contains this pattern is likely to be vulnerable. The reason is that the repetition group is generic enough to contain most of the possible prefixes and the = character guarantees that there exists a failing suffix. For example, the regular expression `/ab[^=]=/` can be exploited using a long string `"abababab.."`.

Given a set of regular expression that match our queries, we manually inspect the context in which the regular expressions are used. The goal is to find matching operations on data that may be delivered through an HTTP request to a web server. To this end, we focus on (i) modules included in the Express framework, (ii) middleware modules that extend this framework, and (iii) modules that manipulate HTTP request components, such as the body or a specific header. For regular expressions in these modules, we keep only those where a data flow from the package interface or from an HTTP header to the regular expression is possible.

To illustrate the size of the search space, consider the number and nature of regular expressions in the most popular 10,000 npm modules. We extract a total of 324,791 regular expressions, with a mean of 63.67, a median of 5.00 and a maximum of 19,791 per module. After removing regular

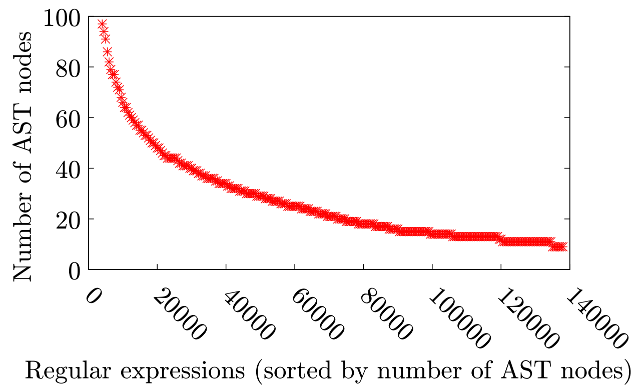


Fig. 5: Complexity of regular expressions that have at least one repetition.

expressions that contain no repetitions, and hence are immune to algorithmic complexity attacks, we obtain a total of 138,123 expressions, with mean 37.93 and median 4.00 per module. To assess the complexity of regular expressions, we parse them into an AST[6] and count the number of AST nodes, shown in Figure 5. More than half of the regular expressions have at least 20 AST nodes, out of which at least one is a repetition. We conclude that the majority of npm packages contains multiple non-trivial regular expressions.

Considering the huge number of potentially vulnerable regular expressions and the manual nature of our analysis, our findings are likely to hugely underapproximate the degree of vulnerability of real-world websites. Overall, it took one of the authors only a couple of days to find 25 vulnerabilities in widely used npm modules, showing that a skilled individual can attack real-world websites with moderate effort. A more powerful attacker could easily detect a larger number of vulnerabilities and perform a larger-scale attack. Future work on automated tools for detecting ReDoS vulnerabilities will further ease this process.

C. *Creating Exploits*

Based on the ReDoS vulnerabilities in npm modules, we create exploits targeted at web servers that use these modules. The main idea is to hypothesize how a server-side web application might use a module. To this end, we set up a fresh Express installation and implement an example web application that uses the module. For example, for a package that parses the user agent, we build an application that parses the user agent of every HTTP request for the main page, which might be used to track the visitors of the website. Next, we try to create an HTTP request where user-controlled data reaches the vulnerable regular expression, and craft input values that trigger an unusually long matching time. For crafting the input, we try to confuse the regular expression engine by forcing it to backtrack because the input can be matched in multiple ways [18], [39]. While creating exploits, we assume that the maximum header size is 81,750 characters, which is the default in Express.js. If we succeed in crafting an input that takes more than five seconds, we consider the vulnerability as exploitable and consider it for the remainder of the study.

The careful reader may notice that our approach does not guarantee that the regular expression has super-linear matching time. However, from a security point of view, having a server that replies to only one request in five seconds is a serious threat to availability.

To further assess the impact of the exploits, we measure how much longer it takes to process a crafted input compared to a random string of the same length. We use two ways of measuring the time. First, we measure at the module boundary, called *matching time*. We measure at module level and not at regular expression level, because this measure may include the effect of multiple regular expressions used in the same module or of other time consuming activities. Second, we measure the time of an entire HTTP request, called *response time*. The response time may include various other components such as: HTTP parsing and serialization, DNS resolving, routing time for the package, HTTP retransmissions, package fragmentation, etc. To measure the response time of a site, we request its main page. For complex sites, this measure underapproximates the time a human user needs to wait for the page to load, because complex sites require separate requests for images etc.

D. *ReDoS Analysis of Websites*

Given a set of exploits, the next step is to measure how many websites are vulnerable to a ReDoS based on one of the exploits. The main challenge is to draw meaningful conclusions about the harm that an attacker *could* cause, without actually attacking live websites in a way that causes harm. During our initial experiments we sent one, not very large request with a crafted header and it appeared to make the analyzed website unresponsive for almost a minute. The goal of our methodology is to avoid this type of mistake.

We address this challenge by triggering requests with increasing input sizes, using both malicious and random inputs, while measuring the response times. Based on locally performed experiments, we choose input sizes that are unlikely to block the server for more than a small, configurable amount of time (we use two seconds in our experiments). If the response time for crafted input grows faster than with random inputs, then we successfully influenced the CPU time consumed on the server and classify the website as exploitable.

The effectiveness of our exploitability test relies on how to measure response time in a reliable way. This is a non-trivial task because of DNS resolving, network caching, delays, retransmissions, and other influencing factors. Another issue is how to determine whether one response time is larger than another in a statistically reliable way. We address these issues (i) by repeating a request three times to "warm up" the connection (e.g., to fill network caches), (ii) by then repeating the request another five times while recording the response times, and (iii) by finally comparing response times from

[6]ASTs of regular expression, as provided by regulex npm module.

random and crafted inputs with tests for statistical significance. If and only if the random and crafted response times have a statistically significant difference and this difference increases when the input size increases, then we classify the site as exploitable.

More formally, suppose we consider k input sizes that each yield two sets T_{random} and T_{attack} of time measurements. For each input size, we compare the confidence intervals of the values in T_{random} and T_{attack} and conclude that the response times differ if the intervals do not overlap. If the response times differ for all input sizes, we quantify the difference for an input size as the difference between $\overline{T}_{random}$ and $\overline{T}_{attack}$, where $\overline{T}$ is the average of the times in T. For k input sizes, this comparison gives a sequence of differences $d_1, .., d_k$. We finally consider a site to be exploitable if $d_1 < d_2 < .. < d_k$.

To obtain time measurements for a site that may suffer from a vulnerability, we need to pick a sequence of input sizes. The challenge is to measure a difference when there is one without repeatedly causing the server to block for longer than a few seconds. We address this challenge through experiments on a locally installed version of the vulnerable package. We craft input sizes that take approximately 100ms, 200ms, 500ms, 1s and 2s to respond to. Considering that we run each payload eight times (three for warmup and five for measuring), this setup blocks the main thread of a vulnerable site for a total of about 30 seconds. We believe this is an acceptable value, since most of the websites have some type of redundancy and blocking one server instance does not block the whole website. However, if the payloads are sent by an attacker in a larger number, they can cause real and severe harm to vulnerable sites, as we show in Section IV-C.

E. Analysis of Mitigation Techniques

Some sites reject requests with large headers and instead return a "400 Bad Request" error. This mitigation can limit the damage of ReDoS attacks. To measure whether a site uses this mitigation technique, we create benign requests of different sizes and measure how often a site reject the request.

IV. Results

This section presents the results of applying the methodology described in Section III to live, real websites. We perform our measurements using three different machines depending on the experiments: a ThinkPad 440s laptop with four Intel i7 CPUs and 12GB memory (Section IV-A), a third party commercial web server with 512MB memory (Section IV-C and IV-D) and a server with 48 Intel Xeon CPUs and 64GB memory (from Section IV-F on).

A. Vulnerabilities and Exploits

Figure 6 shows the modules for which we found at least one vulnerable regular expression that can be exploited through the module's interface. Each vulnerability is working on the latest available version of the package which we show in the second column. The packages vary in the number of dependencies and downloads, but we can safely conclude that ReDoS vulnerabilities are present even in the most popular packages.

Module	Version	Number of dependencies	Downloads in July 2017
debug	2.6.8	16,055	54,885,335
lodash	4.17.4	49,305	44,147,504
mime	1.3.6	2,798	22,314,018
ajv	5.2.2	758	17,542,357
tough-cookie	2.3.2	302	15,981,922
fresh	0.5.0	197	14,151,270
moment	2.18.1	14,421	10,102,601
forwarded	0.1.0	31	9,883,630
underscore.string	3.3.4	2,486	7,277,966
ua-parser-js	0.7.14	225	5,332,979
parsejson	0.0.3	19	4,897,928
useragent	2.2.1	191	3,515,292
no-case	2.3.1	18	3,321,043
marked	0.3.6	2,624	3,012,792
content-type-parser	1.0.1	8	2,337,147
platform	1.3.4	128	757,174
timespan	2.3.0	34	523,290
string	3.3.3	911	421,700
content	3.0.5	9	316,083
slug	0.9.1	499	151,004
htmlparser	1.7.7	178	138,563
charset	1.0.0	36	112,001
mobile-detect	1.3.6	101	107,672
ismobilejs	0.4.1	50	44,246
dns-sync	0.1.3	7	10,599

Fig. 6: Modules with at least one previously unknown vulnerability. The emphasized modules are used to analyze real websites because we found an exploit.

Given the amount of possible damage entailed by the vulnerabilities, we have invested significant efforts to disclose them in a responsible way. For each vulnerability, we have contacted the developers either directly or through the Node Security Platform[7], and gave them several months to fix the problem before making it public. 14 of the 25 have been fixed by now and are listed as advisories on the Node Security Platform. For the others, the developers are either still in the process of fixing or decided to leave the task of fixing to the community. The complete list of vulnerabilities, along with details on their current status is available for the reviewers.[8]

For some of the vulnerable packages, there exist reports of ReDoS vulnerabilities detected by others that are fixed by now. Interestingly, we report in this section more ReDoS vulnerabilities than the whole community did in the past: Between March 2013 and August 2017, there were 22 ReDoS vulnerabilities among a total of 363 vulnerabilities. We started to report our 25 vulnerabilities in August 2017.

As explained in Section III-C, we try to create exploits for the vulnerabilities by hypothesizing how web server implementations may use the vulnerable modules. Figure 7 shows the modules and usage scenarios for which we could create

[7]https://nodesecurity.io/advisories

[8]https://docs.google.com/spreadsheets/d/17uTZ0dhBavwu_F6khLwEBDktmRbQIATqnd3tDxLIIbI/edit?usp=sharing

ID	Module	Header	Usage scenario	JavaScript example
1	charset	Content-Type	The website uses this package to parse the content type of every request.	`require("charset")(req.headers);`

2	content	Content-Type	The website uses this package to parse the content type of every request.	`var content = require("content");`
				`content.type(req.headers["content-type"]);`

3	fresh	If-None-Match	The website uses `express`, which by default uses this package to check the freshness of every request.	`var fresh = require("fresh");`
				`fresh(req.headers);`

4	forwarded	X-Forwarded-For	The website uses `express` and the "trust proxy" option is set. This package is then used to check which proxies a request came through.	`var forwarded = require("forwarded");`
				`var addrs = forwarded(req);`

5	mobile-detect	User-Agent	The website uses this package to get information about the requester.	`var MobileDetect = require("mobile-detect");`
				`var md = new MobileDetect(req.headers["user-agent"]);`
				`md.phone();`

6	platform	User-Agent	The website uses this package to get information about the requester.	`var platform = require("platform");`
				`var agent = platform.parse(req.headers["user-agent"]);`

7	ua-parser-js	User-Agent	The website uses this package to get information about the requester.	`var useragent = require("ua-parser-js");`
				`var agent = useragent.parse(req.headers["user-agent"]);`

8	useragent	User-Agent	The website uses this package to get information about the requester.	`var useragent = require("useragent");`
				`var agent = useragent.parse(req.headers["user-agent"]);`

Fig. 7: Usage scenarios for vulnerable modules and the headers we hypothesize the modules to process.

an exploit. For all the scenarios we assume the payload is sent using a specific HTTP header. We believe that HTTP bodies, UDP packages or WebSocket messages can also be used for the same purpose. The last column of Figure 7 shows the JavaScript implementation of the usage scenario. We run these implementation on our local server to experiment with the exploit.

Most of the scenarios and their implementations are relatively simple. This simplicity shows that an attacker that follows a methodology similar to ours could create exploits that might work for a wide range of websites with relatively little effort. For an attack targeted at a specific website, we believe that more complex scenarios could be build, e.g., involving multiple HTTP requests and domain knowledge. For example, the `marked` package provides a parser for the markdown format. By crafting a specific markdown document, an attacker can block the main loop for hours. However, to deploy the exploit, complex interactions with the server are needed. That is, the attacker needs to figure out which part of the website may use a markdown parser and how to provide a document that will be processed by the parser. We believe that such a scenario is realistic, but it requires an in-depth analysis of each website. We leave for future work to test this hypothesis. In this work, our goal is to assess the effect of exploits that can be deployed at a large scale. Therefore, we only consider very simple usage scenarios that can be triggered with a single HTTP request made to the main page.

To better understand the vulnerabilities, Figure 8 shows for each vulnerable module the vulnerable regular expressions. Some of the expressions are non-trivial, making it hard for developers to focus on possible ReDoS attacks in addition to the correctness of the regular expression. Four of these regular

ID	Vulnerable regular expression
1	`/(?:charset|encoding)\s*=\s*['"]? *([\w\-]+)/i`

2	`/^([^\/]+\/[^\s;]+)(?:(?:\s*;\s*boundary=`
	`(?:"([^"]+)"|([^;"]+)))|(?:\s*;\s*[^=]+=`
	`(?:(?:"(?:[^"]+)")|(?:[^;"]+))))*$/i`

3	`/ *, */`
4	`/ *, */`

5	`new RegExp("Dell.*Streak|Dell.*Aero|Dell.*Venue`
	`|DELL.*Venue Pro|Dell Flash|Dell Smoke|Dell Mini`
	`3iX|XCD28|XCD35|\\b001DL\\b|\\b101DL\\b|\\bGS01\\b")`

6	`/^ +| +$/g`
7	`/ip[honead]+(?:.*os\s([\w]+)*\slike\smac|;\sopera)/`

8	`/((?:[A-z0-9]+|[A-z\-]+ ?)?(?: the)?`
	`(?:[Ss][Pp][Ii][Dd][Ee][Rr]|[Ss]crape|[A-Za-z0-9-]*`
	`(?:[^C][^Uu])[Bb]ot|[Cc][Rr][Aa][Ww][Ll])[A-z0-9]*)`
	`(?:(?:[ \/]| v)(\d+)(?:\.(\d+)(?:\.(\d+))?)?)?/`

Fig. 8: Vulnerable regular expressions.

expressions can be successfully identified by a recent approach proposed by Wüstholz et al. [39], which targets Java applications, though. The remaining four regular express, i.e., IDs 3, 4, 5, and 6, belong to a class of ReDoS-vulnerable regular expressions neglected that, to the best of our knowledge, has so far been neglected in the literature. Instead of having two repetitions that confuse each other, they have only one explicit repetition. For instance, in `/^ +| +$/`, the second repetition is implied by the fact that the regular expression is used in a subgroup matching fashion. More precisely a long string of spaces followed by a non-space character will make the algorithm attempt to match the string starting from position 1, 2, etc. until the end, when the non-space character causes the matching to fail. This type of vulnerability is also special in the sense that it does not require a backtracking step at the

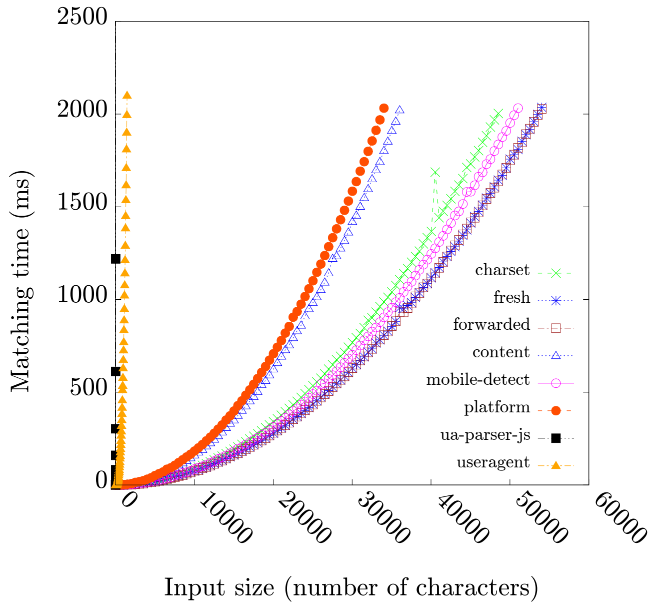


Fig. 9: Matching time of the payloads for different input sizes.

state machine level. However, as we show later in this section, they also cause super-linear complexity and make websites vulnerable to ReDoS.

B. *Matching Time*

We use the exploits to measure the influence of the size of the input to the matching time of the vulnerable expression. Our exploits are not necessary optimal for a given regular expression, nor are they the only ones that can be created. We took a pragmatic approach, as described in Section III, and consider everything that blocks the main thread for multiple seconds on our machine an exploit. Figure 9 shows the matching time as a function of the input size. The measurements use three warm up runs and five actual measurement runs, to account for JIT optimizations and other JavaScript engine-level effects. We then average the results across the five measurement runs to obtain a point on the graph. For most of the exploits, the input dependency seem to be quadratic, reaching one second of matching time within 20,000 to 40,000 characters. For two exploits, however, the input dependency is presumably exponential, reaching 1 second matching time with less than 1,000 characters. We consider any of these eight exploits to be potentially harmful. See Section V-A for a discussion on how non-exponential ReDoS vulnerabilities may aid an attacker in mounting DoS attacks and Section IV-C on what is their actual impact on availability.

To further illustrate the effectiveness of inputs crafted for a specific regular expression, we measure the matching time for each vulnerable module with randomly created inputs. It turns out that random string inputs of the same size as our crafted exploits cause much lower matching times. The maximum matching time across the eight attacks is 20 milliseconds for inputs with 100,000 characters. We conclude that crafting inputs for vulnerable regular expressions is significantly more effective, from an attacker's perspective, than launching a brute-force DoS attack with randomly created inputs.

C. *Availability*

We now show that the matching time of a regular expression has a direct impact on the availability of a web server. To show the threat to availability posed by ReDoS exploits, we create a simple Express application with two features: it replies with a "hello world" message when called at the "/echo" path, and it calls the `forwarded` module with the request headers when called at the "/redos" path. We choose this module because it appears in Figure 9 to be the *least* harmful in our set of exploits, i.e., we are underestimating the negative impact on availability. We then upload this simple application on a machine running Node.js, provided by a commercial cloud platform[9].

We set up two other machines to concurrently send request. One machine, called the victim, measures the time it takes to trigger 100 requests of the "hello world" message. This victim machine triggers the next request once the previous request has been responses to. At the same time, the other machine, called the attacker, delivers 1,000 ReDoS payloads, by triggering all 1,000 requests at once. The victim machine starts its requests immediately after the victim machine has triggered its requests.

We vary the payload size from 0 characters to 8,000 characters in increments of 1,000 characters. A zero-sized payload is a request with an empty header instead of one that exploits the ReDoS vulnerability. We consider the zero-sized payload to check whether a Node.js server can be blocked using a brute-force strategy. We chose the upper limit for the payload size because, by default, the web server provider limits the size of the header fields to 8,500 characters. Other hosting providers allow significantly larger headers, as we report later in this section.

Figure 10 shows the response times measured at the victim machine for the first 25 "/echo" requests. Payloads smaller than 4,000 characters have no significant effect on the response time of the server. In contract, payloads larger than this value delay as many as eight requests with a maximum delay of 20 seconds. By increasing the size of payloads, an attacker can control both the number of requests we delay and their duration. For the largest payloads we use, we even experienced dropping of requests.

This result is particularly remarkable because an individual payload of size 4,000 does not require an immense about of time to respond to. We separately measured the CPU time required to respond to one such request and find it to take only 5.73 milliseconds, on average. However, several requests together can delay the victim's request by up to 20 seconds. This finding shows that the ReDoS payloads have a cumulative effect and even a small delay in the main loop can cause significant harm for availability.

[9]http://heroku.com

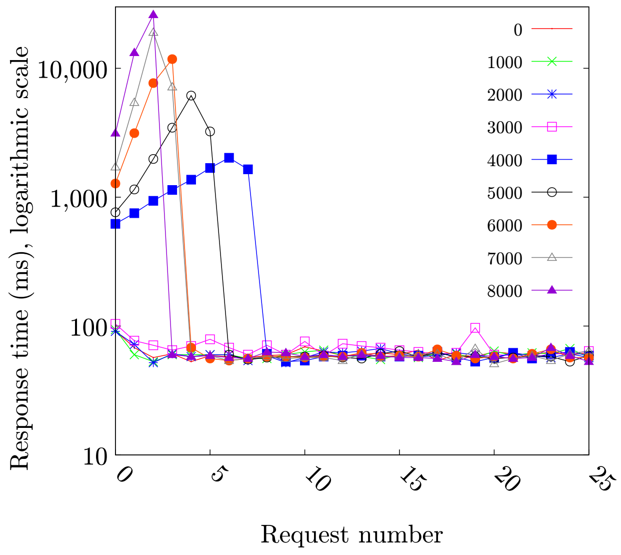


Fig. 10: Impact of differently sized payloads on a server's response time. Note the logarithmic y-scale. The payload sizes are plotted in increments of 1,000 characters.

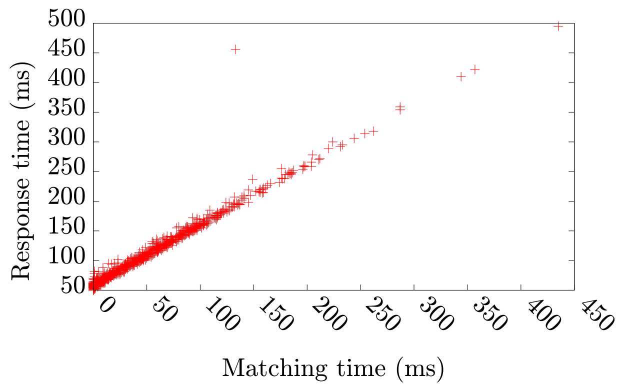


Fig. 11: Correlation between server computation time and request response time.

We remind the reader that the above experiment uses the smallest payload in our data set, `forwarded`. Therefore, if we show that even this exploit poses a threat to availability, we can conclude that the rest of the exploits also do. For more severe ReDoS vulnerabilities, e.g. in `ua-parser-js`, there is even no need to evaluate the impact on availability. As described in the Section II, one single such payload is enough to completely block the server for as long as the matching takes. Considering that with 50–60 characters we predict a CPU computation time in the order of years, such vulnerabilities are a very serious threat to availability.

D. Response Time vs. Matching Time

Our methodology relies on the assumption that small changes in the server computation time have an effect on clients. To validate this assumption we again use the `forwarded` package and the commercial web server setup from the previous section. We use 1,000 payloads smaller than 8,000 characters. The largest one of these payloads produces a matching time smaller than 100 milliseconds on our local machine. We measure the time spent by the server in the `forwarded` package and the time it takes for a request to be served at the client level. We then plot the relation between these two time measurements in Figure 11. The correlation between both measurements is 0.99, i.e., very strong. The strong correlation shows that the delays introduced by the network layer are relatively constant over time and that the server computation time is the dominant component in the response time measured at the client-side. Of course, the observed value depends on the chosen web server provider and the current server load, but we can safely conclude that measuring time at the client level is a good enough estimation of the server-side computation time.

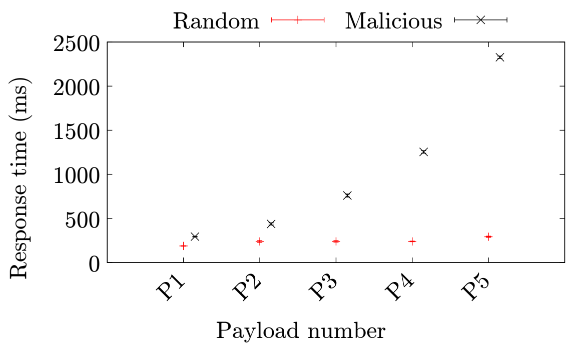
Module	P1: 100ms	P2: 200ms	P3: 500ms	P4: 1s	P5: 2s
fresh	12,000	17,000	27,000	37,500	53,500
forwarded	12,000	17,000	26,500	38,000	53,500
useragent	500	650	925	1,150	1,450
ua-parser-js	38	39	40	41	42
mobile-detect	10,500	15,500	25,000	36,500	50,500
platform	7,500	11,000	17,500	25,000	34,500
charset	10,500	15,500	24,000	34,000	48,000
content	8,000	11,000	18,000	25,500	35,500

Fig. 12: The number of characters in each payload needed to achieve a specific delay in a vulnerable module.

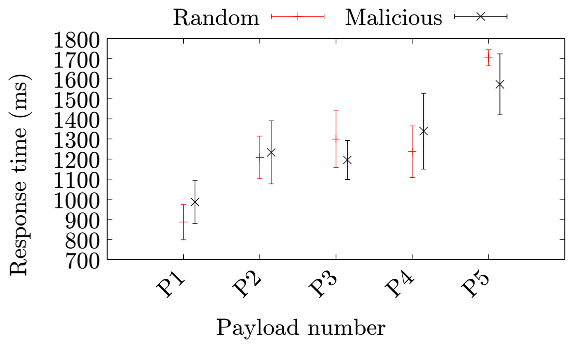
E. Dimensioning Exploits

Choosing an appropriate size for the payload is a crucial part in our methodology and distinguishes our study from a real DoS attack on websites. The goal of this step is to find a payload size that is large enough to check whether a website is vulnerable to a specific attack, but small enough to only block the website for a negligible amount of time. To this end, we locally run each exploit five times with a payload of increasing size and stop the process when the matching time exceeds two seconds. We consider five target matching times, 100ms, 200ms, 500ms, 1s and 2s, and choose the payload size that produces the closest matching time to the target time.

Figure 12 shows the values for each target time and vulnerable module. For example, we estimate that for the `platform` vulnerability, we obtain a matching time of 200ms with a payload of 11,000 characters. It is worth highlighting the `useragent` and `ua-parser-js` packages, whose matching times grow at a much faster rate, requiring less than 1,500 characters to cause a delay of 2s.



(a) Response time for an vulnerable site.



(b) Response time for a non-vulnerable site.

Fig. 13: Effect of increasing payload sizes on the response time of two websites.

Exploit	Affect sites
fresh	241
forwarded	99
ua-parser-js	41
useragent	16
mobile-detect	9
platform	8
charset	3
content	0

Fig. 14: Number of websites affected by specific vulnerabilities.

F. Vulnerable Sites

The goal of the next step is to assess to what extent real websites suffer from ReDoS vulnerabilities. Based on the five payload sizes for each exploit, we create attack payloads and random payloads for each exploit and payload size. We send these payloads to the 2,846 real websites that are running an Express webserver (Section III-A). We warm up the connection three times and then measure five response times for both random and malicious inputs. Using the methodology described in Section III-D, we then decide based on the measured response times whether a site is vulnerable. If for some reason, we could not send three or more out of the five payloads to a specific website, we consider that website to be non-vulnerable.

Overall, we observe that 339 sites suffer from at least one of the eight vulnerabilities. 66 sites actually suffer from two vulnerabilities and six sites even from three. This result shows that ReDoS attacks are a widespread problem that affects a large number of real-world websites. Given that our methodology is designed to underestimate the number of affected sites, e.g., because we consider only eight exploits, the actual number of ReDoS-vulnerable sites is likely to be even higher. Moreover, we expect the growing popularity of JavaScript on the server side to further increase the problem in the future.

To illustrate our methodology for deciding whether a site is vulnerable, consider two example websites. In Figure 13, we plot for each of the five payload sizes the response time for malicious and random inputs. The figure shows the mean and the confidence intervals for a vulnerable site in Figure 13a and for a non-vulnerable site in Figure 13b. The response time grows significantly faster for the malicious payloads in the vulnerable site, reaching slightly more than two seconds for the fifth payload. In contrast, for the non-vulnerable site, the response time for both malicious and random payloads seems to grow linearly. Since the confidence interval for the response times in Figure 13b overlap, we classify this website as non-vulnerable. By inspecting other websites classified as vulnerable by our methodology, we observe patterns similar to Figure 13a. Therefore, we conclude that our criteria for deciding if a website is vulnerable are valid.

G. Prevalence of Specific Vulnerabilities

Figure 14 shows the number of websites affected by each vulnerability. Perhaps unsurprisingly, the vulnerabilities in `fresh` and `forwarded` have most impact, since these two modules are part of the Express framework. One of them needs to be activated using a configuration option, while the other module is enabled by default. One may ask why not all Express analyzed websites suffer from this problem. The reason is the way we dimension our payloads: Many Express instances limit the header size, and hence we cannot send large enough payloads to confirm that the sites are vulnerable. The other six vulnerabilities affect websites with a frequency that is roughly proportional to the popularity of the respective modules. For example, the vulnerability in the popular `useragent` affects more websites than the vulnerability in the less used `charset` module. To our initial surprise, we cannot confirm any site vulnerable due to the `content` module. After more careful consideration, we realized that there are two more popular alternatives for parsing the `Content-Header` and the `content` package seems to be more popular among users of the hampi framework, which is a competitor of Express.

From an attacker's perspective, the distribution of vulnerabilities is great news, because exploits are portable across websites and knowing a vulnerabilities is sufficient to attack various websites. Likewise, the distribution is also good news

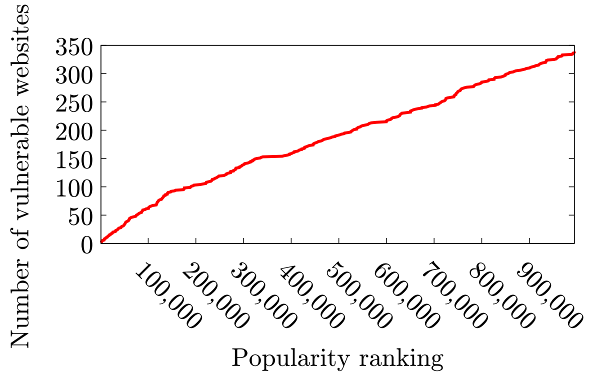


Fig. 15: Cumulative distribution function showing the popularity of vulnerable sites. Each point on the graph shows how many sites among the top x sites suffer from at least one vulnerability.

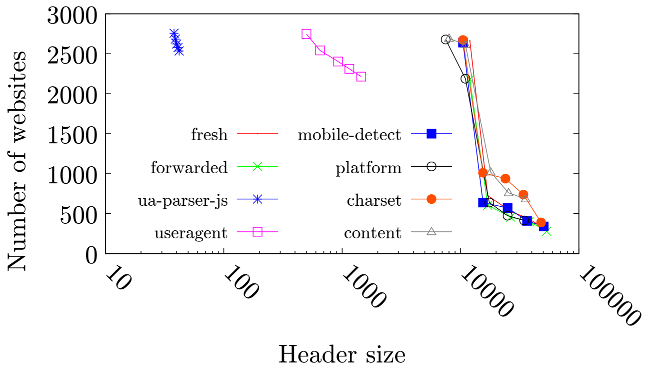


Fig. 16: Number of websites that accept a payload of a specific size. Note the logarithmic x-scale.

for the community, showing that one can lower the risk of ReDoS in multiple websites by fixing a relatively small set of popular packages.

H. Influence of Popularity

Are ReDoS vulnerabilities a problem of less popular sites? In Figure 15, we show how the vulnerable sites are distributed across the Alexa top one million sites. For each point p on the horizontal axis, the vertical axis shows the number of exploitable sites with popularity rank $\leq p$. For example, there are 61 vulnerable sites in the top 100,000 websites. As can be observed from the distribution, the vulnerabilities are roughly equally distributed among the top one million sites. There is even a slight tendency toward more vulnerabilities among the more popular websites. This tendency can be explained by the trend we have seen in Figure 4, that server-side JavaScript tends to be more popular among popular websites. Overall, we can conclude that ReDoS vulnerabilities are a general problem that affects sites independent of their popularity ranking.

I. Use of Mitigation Techniques

As mentioned before, some websites refuse to process a request whose header size exceeds a certain size. In Figure 16 we plot for each exploit how many websites accept a payload of a given size. As can be observed, most websites accept headers that are smaller than 10,000 characters, but only few websites accept headers that are, for instance, 40,000 characters long. As we have shown in Section IV-C, 10,000 characters are enough to do harm even with the least serious vulnerability. Therefore, the current limits that the websites apply on the header size are insufficient and they do not provide adequate protection against DoS.

Another interesting trend to observe in Figure 16 is that even for the most harmful exploit, `useragent`, for which we require payloads between 38 and 42 characters only, the number of websites that accept larger payloads decreases over time. This is surprising since for other exploits like `mobile-detect` there seem to be more websites to accept 10,000 characters long headers. We believe this observation to be due to the fact that some websites refuse to process many requests from the same user in a short period of time. For instance, our largest payload is sent after approximately 50 other requests of smaller size and the site refuses to serve it. This is a well known network-level protection against DoS, but there seem to be only around 200 websites to implement it. However, limiting the number of requests is no silver bullet against denial of service attacks, especially when the attacker has the resources to deploy a distributed denial of service attack.

J. Threats to Validity

One threat to validity for our study is that we rely on time measurements performed over the network to estimate the likelihood of a ReDoS vulnerability. One may argue that these measurements should not be trusted and that pure chance made us observe some larger slowdowns for malicious payloads. We address this threat in multiple ways: we show that for commercial web hosting servers there is a high correlation between response time and server CPU time, we repeat measurements multiple times, and we draw conclusions only from statistically significant differences.

Another potential concern is that the exploits we created are too generic and happen to cause slowdown in another regular expression than the one we created them for. We believe that this situation would only impact our ability to tell which module is used on the server-side and not the impact of a ReDoS attack. Moreover, five of our exploits rely on a specific sequence of characters in the payload to the effective. These sequences of highly contextual characters need to be present in the beginning or at the end of the exploit. Removing any of them would make the exploit unusable. Therefore, we believe that at least for these vulnerabilities it is very likely that our exploits indeed trigger the intended regular expression.

V. Discussion

In this section, we discuss the potential of a large-scale DoS attack on Node.js websites and some defenses we recommend to minimize the impact of such an event. Finally, we describe an unexpected implication of our study: that algorithmic complexity attacks can be used for software fingerprinting.

A. *Impact of a Large-scale Attack*

Compared to a regular DoS attack, a ReDoS vulnerability enables an attacker to launch an attack with fewer resources. As shown in Section IV-C, even the least harmful vulnerabilities we identify can be a lethal weapon when used as part of a large-scale DoS attack. More precisely, instead of hanging the event loop for few milliseconds with a benign request, the attacker can send payloads that hang the loop for hundreds of milliseconds, several seconds, or even more, depending on the vulnerability. We remind the reader that with just eight standard attack vectors we could affect hundreds of websites.

It is worth emphasizing once again that this issue would not be as serious in a traditional thread-based web server, such as Apache. This is because the matching would be done in a thread serving the individual client. In contract, in an event-based system, the matching is done in the main loop and spending a few seconds matching a regular expression is equivalent to completely blocking the server for this amount of time.

A large-scale ReDoS attack against Node.js-based sites is a bleak scenario for which, as we have shown, many websites are not prepared. To limit this risk, we have been working with the maintainers of vulnerable modules to fix vulnerabilities. In addition, we urgently call for the adoption of multiple layers of defense, as outlined in the following.

B. *Defenses*

First of all, to limit the effect of a payload delivered through an HTTP header, the size of the header should be limited. For more than 15% sites, we could successfully deliver headers longer than 25,000 characters. We are not aware of any benign use cases for such large HTTP headers. Therefore, a best practice in Node.js applications should be to limit the size of request headers. This kind of defense would mitigate the effects of some potential attacks, but is limited to vulnerabilities related to HTTP headers. In contrast, vulnerabilities related to other inputs received from the network, e.g., the body of an HTTP request, would remain exploitable.

Another defense mechanism could be to use a more sophisticated regular expression engine that guarantees linear matching time. The problem is that these engines do not support advanced regular expression features, such as lookahead or back-references. Davis et al. [11] advocate for a hybrid solution that only calls the backtracking engine when such advanced features are used, and to use a linear time algorithm in all other cases. This is an elegant solution that is already adopted by languages like Rust[10]. However, it would not completely solve the problem, since some regular expressions with advanced features may still contain ReDoS vulnerabilities. For instance, during our vulnerability study, we found the following regular expression:

```
/(?=.*\bAndroid\b)(?=.*\bMobile\b)/i
```

This expression from the `ismobilejs` module contains both lookahead and has super-linear complexity in a backtracking engine.

We also recommend that Node.js augments its regular expression APIs with an additional, optional timeout parameter. Node.js will stop any matching of regular expressions that takes longer than the specified timeout. This solution is far from perfect, but it is relatively easy to implement and adopt, and it has been successfully deployed in other programming languages [22].

Additionally, we advocate that our work should be used as a roadmap for penetration testing sessions performed on Node.js websites. First, the tester audits the list of package dependencies, identifies any known ReDoS vulnerability in these packages or analyzes all the contained regular expressions. Second, the tester creates payloads for all the vulnerable regular expressions identified in the first step. Third, the tester tries to deliver these payloads using standard HTTP requests.

Finally, better tools and techniques should be created to help developers reason about ReDoS vulnerabilities in server-side JavaScript. Both static and dynamic analysis tools can aid in understanding the complexity of regular expressions and their performance. A good starting point could be porting existing solutions that were created for other languages, e.g. [39].

C. *Fingerprinting Web Servers*

Part of our methodology could be used to fingerprint web servers to predict some of the third-party modules used by a website. This ability can be useful for an attacker in at least two ways. First, the attacker may try to temper with the development process of that module by introducing backdoors that can then be exploited in the live website. Given that npm modules often depend on several others, the vulnerability can even be hidden in a dependent module. Second, the attacker may exploit a more serious vulnerability present in the same module. To show how this scenario may happen, consider the `dns-sync` vulnerability, identified in Section IV-A. The vulnerable function suffers both from a ReDoS attack and a command injection attack [33]. An attacker may use the ReDoS attack as a hard-to-detect way to scan which sites use the vulnerable module and then attack these sites with a command injection.

VI. Related Work

a) Server-side JavaScript: Ojamaa and Düüna [24] were the first to asses the general security of Node.js and identified algorithmic complexity attacks as one of the main threats. More recently Davis et al. [11] raise another alarm, showing that ReDoS vulnerabilities are present in popular modules. They even go so far as to claim that algorithmic

[10]https://github.com/rust-lang/regex

complexity attacks are a weakness in the Node.js design and that the community is not ready to handle them. We take these observations further and show that indeed ReDoS is a widespread problem that affects real websites. Other security related studies on Node.js explore topics like command injection vulnerabilities [?] and configuration errors [28]. There are also several techniques proposed for handling Node.js related issues: static analysis that can handle Node.js specific events [23], fuzzing that help uncover concurrency related bugs [12], auto-sanitization that helps protect against injections [33], understanding the event interactions that happen between server-side and client-side code [1]. To the best of our knowledge, our work is the first to analyze Node.js security related problems in real world websites and to demonstrate how an attacker may exploit vulnerabilities in npm modules to attack websites.

b) Analysis of ReDoS Vulnerabilities: There is a lot of interest in analyzing the worst case matching time of regular expressions [6], [37], [18], [2]. Most of this work assumes a backtracking style matching engine and analyzes individual regular expressions. However, a vulnerable regular expression is only dangerous when attacker controlled input flows into it. The only work that we are aware of that takes this aspect into consideration is proposed by Wüstholz et al. [39]. They find 41 vulnerabilities in Java software by using a combination of static analysis and exploit generation in Java software. Our work is different in multiple ways: (i) it analyzes JavaScript ReDoS which is more serious than Java ReDoS, (ii) it detects vulnerabilities in real world websites whose source code is not available for analysis and (iii) it uncovers ReDoS vulnerabilities containing advanced features like lookahead, which are not supported by any of the previous work.

c) Regular Expressions: One important use for regular expressions in the security domain is the implementation of sanitizers and XSS filters. Bates et al. [5] show that XSS filters are often slow or incorrect, and sometimes they even introduce new vulnerabilities. Hooimeijer et al. [15] formally analyze real life sanitizers and show that multiple supposedly equivalent implementations differ on at least some inputs. A recent study by Chapman et al. [9] show that developers have difficulties in composing and reading regular expressions. Our work also shows that developers often get the regular expressions wrong, but we are the first to analyze the impact of this problem in the case of real world websites. A very promising direction for avoiding mistakes in regular expressions is to not write them in the first place, and let a machine synthesize them instead [3], [4].

d) Algorithmic Complexity Attacks: The difference between average and worst case performance can be used to perform a denial of service attack called algorithmic complexity attack. Crosby and Wallach [10] analyze vulnerabilities that enable attacks due to performance of hash tables and binary trees, while Dietrich et al. [13] study serialization-related attacks. Wise [7], SlowFuzz [25], and PerfSyn [35] automatically generate inputs to trigger unexpectedly high complexity. Our work analyzes ReDoS which is a particular class of algorithmic complexity attacks that leverages the limits of the traditional regular expression matching algorithms.

e) Resource Exhaustion Attacks: SAFER [8] statically detects CPU and stack exhaustion vulnerabilities involving recursive calls and loops. Huang et al. [16] study blocking operations in the Android system that can force the OS to reboot when called multiple times. Shan et al. [31] consider attacks on n-tier web applications and model them using a queueing network model. None of these target ReDoS, which is the focus of this paper.

f) Testing Regular Expressions: The problem of generating inputs for regular expressions is also investigated from a software testing perspective [36], [21], [19], [30]. However, compare to ours, this direction has very different objectives for inputs generation, i.e. maximizing coverage or finding bugs in the implementation.

g) Performance of JavaScript: ReDoS vulnerabilities are a kind of performance problem. Such problems are worth fixing independent of their exploitability in a denial of service attack, e.g., to prevent websites from being perceived as slow and unresponsive. Existing work has studied JavaScript performance issues [29] and proposed profiling techniques to identify them [26], [14], [17]. In contrast to that line of work, this paper addresses a security-related performance problem. Studying the exploitability of other performance issues beyond ReDoS is a promising direction for future work.

h) Studies of the Web: Others have also performed large-scale studies of security issues in the web. Lauinger et al. [20] study the use of client-side JavaScript libraries that are outdated and have known vulnerabilities. In contrast to their setup, we focus on ReDoS issues, on server-side code, and on code that is vulnerable despite being up-to-date. Another study looks into attack vectors and defenses related to the `postMessage` API in HTML5 [32], showing that attackers may use it to circumvent the same-origin policy. A study by Richards et al. [27] analyzes the use of JavaScript's `eval` function, which is prone to code injections. All the above studies are orthogonal to our work. To the best of our knowledge, we are the first to focus on server-side JavaScript and on ReDoS vulnerabilities.

VII. Conclusions

This paper studies ReDoS vulnerabilities in JavaScript-based web servers and shows that they are an important problem that affects various popular websites. We exploit eight vulnerabilities that affect at least 339 popular websites. We show that an attacker could block these vulnerable sites for several seconds and sometimes even much longer. More generally, our results are a call-to-arms to address the current lack of tools for analyzing ReDoS vulnerabilities in JavaScript.

References

[1] Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. Understanding asynchronous interactions in full-stack JavaScript. In *Proceedings of the 38th International Conference on Software Engineering, ICSE*, 2016.

[2] Arturs Backurs and Piotr Indyk. Which regular expression patterns are hard to match? In *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS*, 2016.

[3] Alberto Bartoli, Giorgio Davanzo, Andrea De Lorenzo, Eric Medvet, and Enrico Sorio. Automatic synthesis of regular expressions from examples. *IEEE Computer*, 47(12):72–80, 2014.

[4] Alberto Bartoli, Andrea De Lorenzo, Eric Medvet, and Fabiano Tarlao. Can a machine replace humans in building regular expressions? A case study. *IEEE Intelligent Systems*, 2016.

[5] Daniel Bates, Adam Barth, and Collin Jackson. Regular expressions considered harmful in client-side XSS filters. In *Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, April 26-30, 2010*, pages 91–100, 2010.

[6] Martin Berglund, Frank Drewes, and Brink van der Merwe. Analyzing catastrophic backtracking behavior in practical regular expression matching. In *Proceedings 14th International Conference on Automata and Formal Languages, AFL 2014, Szeged, Hungary, May 27-29, 2014.*, pages 109–123, 2014.

[7] Jacob Burnim, Sudeep Juvekar, and Koushik Sen. WISE: Automated test generation for worst-case complexity. In *ICSE*, pages 463–473. IEEE, 2009.

[8] Richard M. Chang, Guofei Jiang, Franjo Ivancic, Sriram Sankaranarayanan, and Vitaly Shmatikov. Inputs of coma: Static detection of denial-of-service vulnerabilities. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium, CSF 2009, Port Jefferson, New York, USA, July 8-10, 2009*, pages 186–199, 2009.

[9] Carl Chapman and Kathryn T. Stolee. Exploring regular expression usage and context in Python. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA*, 2016.

[10] Scott A. Crosby and Dan S. Wallach. Denial of service via algorithmic complexity attacks. In *Proceedings of the 12th USENIX Security Symposium*, 2003.

[11] James Davis, Gregor Kildow, and Dongyoon Lee. The case of the poisoned event handler: Weaknesses in the Node.js event-driven architecture. In *Proceedings of the 10th European Workshop on Systems Security, EUROSEC*, 2017.

[12] James Davis, Arun Thekumparampil, and Dongyoon Lee. Node.fz: Fuzzing the server-side event-driven architecture. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, pages 145–160, 2017.

[13] Jens Dietrich, Kamil Jezek, Shawn Rasheed, Amjed Tahir, and Alex Potanin. Evil pickles: DoS attacks based on object-graph engineering. In *31st European Conference on Object-Oriented Programming, ECOOP*, 2017.

[14] Liang Gong, Michael Pradel, and Koushik Sen. JITProf: Pinpointing JIT-unfriendly JavaScript code. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 357–368, 2015.

[15] Pieter Hooimeijer, Benjamin Livshits, David Molnar, Prateek Saxena, and Margus Veanes. Fast and precise sanitizer analysis with BEK. In *USENIX Security Symposium*, pages 1–16, August 2011.

[16] Heqing Huang, Sencun Zhu, Kai Chen, and Peng Liu. From system services freezing to system server shutdown in Android: All you need is a loop in an app. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 1236–1247, 2015.

[17] Simon Holm Jensen, Manu Sridharan, Koushik Sen, and Satish Chandra. MemInsight: platform-independent memory debugging for JavaScript. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 345–356, 2015.

[18] James Kirrage, Asiri Rathnayake, and Hayo Thielecke. Static analysis for regular expression denial-of-service attacks. In *Network and System Security - 7th International Conference, NSS 2013, Madrid, Spain, June 3-4, 2013. Proceedings*, pages 135–148, 2013.

[19] Eric Larson and Anna Kirk. Generating evil test strings for regular expressions. In *IEEE International Conference on Software Testing, Verification and Validation, ICST 2016*, 2016.

[20] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. Thou shalt not depend on me: Analysing the use of outdated JavaScript libraries on the web. In *NDSS*, 2017.

[21] Nuo Li, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. Reggae: Automated test generation for programs using complex regular expressions. In *ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering*, 2009.

[22] Alex Mackey, William Stewart Tulloch, and Mahesh Krishnan. *Introducing. NET 4.5*. Apress, 2012.

[23] Magnus Madsen, Frank Tip, and Ondrej Lhoták. Static analysis of event-driven Node.js JavaScript applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*, 2015.

[24] Andres Ojamaa and Karl Düüna. Assessing the security of Node.js platform. In *International Conference for Internet Technology and Secured Transactions*, 2012.

[25] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 2155–2168, 2017.

[26] Michael Pradel, Parker Schuh, George Necula, and Koushik Sen. EventBreak: Analyzing the responsiveness of user interfaces through performance-guided test generation. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 33–47, 2014.

[27] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The eval that men do - a large-scale study of the use of eval in JavaScript applications. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 52–78, 2011.

[28] Mohammed Sayagh, Noureddine Kerzazi, and Bram Adams. On cross-stack configuration errors. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 255–265, 2017.

[29] Marija Selakovic and Michael Pradel. Performance issues and optimizations in JavaScript: An empirical study. In *International Conference on Software Engineering (ICSE)*, pages 61–72, 2016.

[30] Muzammil Shahbaz, Phil McMinn, and Mark Stevenson. Automated discovery of valid test strings from the web using dynamic regular expressions collation and natural language processing. In *12th International Conference on Quality Software year = 2012*.

[31] Huasong Shan, Qingyang Wang, and Calton Pu. Tail attacks on web applications. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1725–1739, 2017.

[32] Sooel Son and Vitaly Shmatikov. The postman always rings twice: Attacking and defending postmessage in HTML5 websites. In *NDSS*, 2013.

[33] Cristian-Alexandru Staicu, Michael Pradel, and Ben Livshits. Understanding and automatically preventing injection attacks on Node.js. In *NDSS*, 2018.

[34] Ken Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.

[35] Luca Della Toffola, Michael Pradel, and Thomas R. Gross. Synthesizing programs that expose performance bottlenecks. In *CGO*, 2018.

[36] Margus Veanes, Peli de Halleux, and Nikolai Tillmann. Rex: Symbolic regular expression explorer. In *Third International Conference on Software Testing, Verification and Validation, ICST 2010*, 2010.

[37] Nicolaas Weideman, Brink van der Merwe, Martin Berglund, and Bruce Watson. Analyzing matching time behavior of backtracking regular expression matchers by using ambiguity of NFA. In *Implementation and Application of Automata - 21st International Conference, CIAA*, 2016.

[38] Paul Wilton. *Beginning JavaScript*. John Wiley & Sons, 2004.

[39] Valentin Wüstholz, Oswaldo Olivo, Marijn J. H. Heule, and Isil Dillig. Static detection of DoS vulnerabilities in programs that use regular expressions. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS*, 2017.