

✓ Closed

yetingli opened this issue on Oct 11, 2020 · 15 comments · Fixed by #2584

Labels

language-definitions

yetingli commented on Oct 11, 2020 • edited ▼

Hi,

I would like to report 6 ReDoS vulnerabilities in prism (<https://github.com/PrismJS/prism>).

It allows cause a denial of service if highlighting crafted codes.

- The first ReDos

The vulnerable regular expression is "(?:%s*\n%s*|%,|[^\r\n]*)" and is located in

prism/components/prism-eiffel.js
Line 16 in 38f42dd

```
16 pattern: /"(?:%\\s*\\n\\s*%|%.|["^%\\r\\n]))*/;
```

The ReDOS vulnerability can be exploited with the following crafted code string

```
function aa(){
"%
```

%%

%%

%%

%

7070

7878

99

%%

%%

%%

%%

10/10

7879

000000

0/0/

%%

%%

%%

%

10/10

9/9/

9/9/

%%

%%

I quickly want to point out that the 6th vulnerability doesn't trigger for `'if+'/? '*100+'!`. The reason the pattern is vulnerable is because the first `(?:'["]{0,1}[\\s\\S]*)` is ambiguous and repeated. A working attack string is:

```
if/a: ""/a: ""/a: ""/a: ""/a: ""/a: ""/a: ""/a: ""/a: ""/a: ""/ a defined -
```

RunDevelopment commented on Oct 11, 2020

Member

I fixed 5/6 patterns.

I can't reproduce the 3rd vulnerability. The attack string doesn't work and I just don't see ambiguity that could cause exponential backtracking.

RunDevelopment mentioned this issue on Oct 11, 2020

Fixed multiple cases of vulnerable REs #2584

↗ Merged

yetingli commented on Oct 11, 2020 • edited

Author

I quickly want to point out that the 6th vulnerability doesn't trigger for `'if'+/? '*100+!'`. The reason the pattern is vulnerable is because the first `(?: "[^"]*" | \s+)` ambiguous and repeated. A working attack string is:

```
if/a: ""/a: ""/a: ""/a: ""/a: ""/ a defined -
```

Oops...This is a typo, What I want to write is `'if'+/? "\\"*100+!'`

Yes, your attack string is right, too :)

However, I want to point out that this string `'if'+/? '*100+!'` is also an attack one, because the sub-pattern `(?: \\/[a-z]?([:]"[^\s"]*" | \s+))?` is ambiguous. So the fix for the 6th vulnerability is still incomplete (e.g., the string `'if'+/? '*100+!'` still works).

yetingli commented on Oct 11, 2020

Author

I can't reproduce the 3rd vulnerability. The attack string doesn't work and I just don't see ambiguity that could cause exponential backtracking.

You can make the string longer, for example, the attack string `'= '*50000` took 24763 ms.

The sub-pattern `(?:=+ +)+` is ambiguity, I am willing to suggest that you replace `(?:=+ +)+` with `=+ +`

This fix is equivalent, and the repaired regex is safe and efficient.

For example, for the attack string `'= '*50000`, the vulnerable regex took 24763 ms and the fixed one only took 1 ms.

yetingli commented on Oct 11, 2020

Author

Thank you for reporting! I'll fix them immediately.

@yetingli How did you find the vulnerabilities? Some of the patterns in question contain backreferences (and assertions) and I don't know any existing technique for the static analysis of RE that can handle that. Did you go through the patterns by hand or do you know (I looked at your previous work) did you create a tool that does the analysis?

Yes, I use my tool to detect ReDos vulnerabilities. After a time, I will release my tool and welcome to use it :)

👍 1

yetingli commented on Oct 11, 2020

Author

I fixed 5/6 patterns.

I can help to detect whether the repaired patterns are safe.

yetingli closed this as completed on Oct 11, 2020

yetingli reopened this on Oct 11, 2020

RunDevelopment commented on Oct 12, 2020 • edited

Member

I just understood what you mean by strings like `'if'+/? '*100+!'`. I took them as literal strings (`'"if'+/? '*100+!'"`) and not a python-like expression to generate strings.

I verified and fixed the 6th vulnerability for your attack string.

Yes, I use my tool to detect ReDos vulnerabilities. After a time, I will release my tool and welcome to use it :)

I'm looking forward to it but in that case, I have a question: There are 2 more vulnerabilities in `prism-batch.js` (same file as a 6th one). They are in the regexes in [line 28](#) and [line 62](#). They are vulnerable for the same reason the regexes on line 41 you pointed out is (they all contain the sub pattern `(?: \\/[a-z]?([:]"[^\s"]*" | \s+))?`). Did you intentionally omit this or did your tool fail to detect this?

If your tool failed to detect this because it couldn't extract the regexes from the source code, then I can help you with this. I can modify Prism's test suite to extract all of the 2500 unique regexes Prism uses (even the dynamically generated ones) and output all of them into a JSON file (or similar) (like I did [here](#)). I don't want to push you to spend more time on Prism than necessary, so feel free to decline if you aren't interested.

I can help to detect whether the repaired patterns are safe.

Thank you for your continued help! All of my fixes are in [#2584](#).

I can't reproduce the 3rd vulnerability. The attack string doesn't work and I just don't see ambiguity that could cause exponential backtracking.

You can make the string longer, for example, the attack string `'= '*50000` took 24763 ms.

The sub-pattern `(?:=+ +)++` is ambiguity, I am willing to suggest that you replace `(?:=+ +)++` with `=+ ++`

This fix is equivalent, and the repaired regex is safe and efficient.

For example, for the attack string `' = '*50000`, the vulnerable regex took 24763 ms and the fixed one only took 1 ms.

I analyzed the pattern and the problem is the following:

First of all. The pattern does not backtrack exponentially. All words of the language `' = '*n` (I'm using your notation) are rejected after $O(n^2)$ steps.

► Measurements

That being said, I don't think that this can actually be fixed. The problem is that each suffix of the input string takes $O(n)$ steps to reject and it seems like the regex engine is forced to try $O(n)$ many suffixes. Even extremely simple regexes like `/a+b/` show this $O(n^2)$ behavior for inputs like `'a'*n`.

I also want to point out that `(?:=+ +)++` and `=+ ++` do not accept the same language but `(?:=+ +)++` and `=+ [=]*=` do.

I will replace all occurrences of `(?:=+ +)++` with `=+ [=]*=` for now. While `=+ [=]*=` will still reject your attack string in $O(n^2)$ time, it does so twice as fast as `(?:=+ +)++` in my tests.

yetingli commented on Oct 12, 2020

Author

It's very nice to receive your reply.

Did you intentionally omit this or did your tool fail to detect this?

I want to point out that my tool does not support extracting regular expressions from the projects, but only supports inputting regular expressions to check whether regular expressions are safe.

I can modify Prism's test suite to extract all of the 2500 unique regexes Prism uses (even the dynamically generated ones) and output all of them into a JSON file (or similar) (like I did [here](#)).

I don't want to push you to spend more time on Prism than necessary, so feel free to decline if you aren't interested.

Great! You can send me these regexes and I am willing to check whether all regexes Prism uses are safe.

I also want to point out that `(?:=+ +)++` and `=+ ++` do not accept the same language but `(?:=+ +)++` and `=+ [=]*=` do.

I will replace all occurrences of `(?:=+ +)++` with `=+ [=]*=` for now. While `=+ [=]*=` will still reject your attack string in $O(n^2)$ time, it does so twice as fast as `(?:=+ +)++` in my tests.

Indeed, they are not equivalent and you're right.

RunDevelopment commented on Oct 12, 2020

Member

You can send me these regexes and I am willing to check whether all regexes Prism uses are safe.

Thank you very much! [Here](#) is a JSON of all 2587 unique regexes. Each regex is mapped to all of its occurrences in Prism language definitions. All regexes were extracted from the most recent commit in [#2584](#).

If you need/want anything else, please don't hesitate to ask. This is a huge improvement for Prism, so I'm really thankful for the work you're doing.

This was referenced on Oct 12, 2020

Detect quadratic patterns RunDevelopment/eslint-plugin-clean-regex#23

🔒 Closed

Added test for exponential backtracking #2590

🔗 Merged

yetingli commented on Oct 16, 2020 • edited

Author

I have detected 116 vulnerable regular expressions (see the [link](#), there are some scripts for you to verify further).

Feel free to contact me if you have any questions.

RunDevelopment commented on Oct 25, 2020

Member

Thank you @yetingli!

Sorry for the delay. I have looked at the files and analyzed them.

Changed regexes

Before I show the results, one question: Why did some regexes change?

None of the regexes in your JS files had flags. I went through a few regexes and it seems like none of them actually used the flags but I still wonder. The other change I noticed is more severe: Non-capturing groups were replaced with capturing groups. This is a huge issue because this changed the groups that backreferences were pointing to.

► Example of a changed regex

There are no other changes that have been made to the regexes.

I also want to point out that the second change (non-capturing -> capturing) is not consistent across regexes. The second change is significantly more likely for the regexes with higher file numbers.

Extraction

While the script files were very nice in demonstrating the issue, they also make it hard to fix the issues because I had no idea where the regexes are coming from. The first thing I did was to extract the regex and the `[x,y,z]` tuple (the strings that generate the attack string `x+y*n+z`) from all scripts. I then added the location of each regex (the array of strings each regex had in the JSON I send you).

Measuring

I then measured the execution time of each regex on generated attack strings `x+y*n+z` to determine the runtime complexity of the regexes. This can easily be done because the runtime of a regex will either be linear, polynomial or exponential.

Note: With linear, I mean $O(n)$. With polynomial, I mean $O(n^p)$ with $p \geq 2$.

The runtime of both the extracted regex and the original regex were measured. I also measure the runtime of an anchored version of the regex (`(= /^<extracted regex>/)`) to determine the whether the polynomial runtime is due to the pattern being matched against $O(n)$ suffixes of the string.

Results

5 regexes were changed enough to affect their runtime. This was usually due to a backreference not referring to the right group anymore.

5 regexes have exponential runtime. 4 of those, I already detect (and fixed) with my method in [#2590](#). The remaining one (`Prism.languages.asciidoc.macro`) was fixed.

2 regexes have linear runtime. One attack string didn't work because a prefix of it was accepted. The other attack string didn't work because the regex had Prism lookbehind group (just a normal capturing group) that prevented polynomial runtime.

Polynomial

Now to the polynomial. 104 regexes have polynomial runtime. I further subdivided this into 3 categories. The main idea is that some of the polynomial runtime isn't caused by backtracking but by the regex engine moving the pattern across the string to find a match. Even if the pattern only looks at each character once to reject a suffix of the string, it has to do so for $O(n)$ many times due to being moved. (Example: `a*b` for strings `"a"*n`.) This is why I also measured an anchored version of each regex.

95 regexes have $O(n^2)$ runtime because of moving. This is the majority of all regexes.

1 regex has $O(n^3)$ runtime because of moving and polynomial ($O(n^2)$) backtracking.

8 regexes have polynomial backtracking not affected by moving.

Problems with your method

I want to point out that your method seems to favor polynomial runtime over exponential backtracking. Some of the patterns reported as polynomial also have attack strings that can cause exponential backtracking.

► Examples:

This is concerning because people may be inclined to ignore fixing patterns that run in "only $O(n^2)$ ". This may cause some exponential backtracking to never be found and fixed.

I also want to point out that your method has failed to point out a lot of other patterns with exponential backtracking.

► Examples:

It also failed to find a lot of cases of polynomial backtracking. See [#2597](#) for examples.

Closing thoughts

While not perfect, your method has found a lot of cases of exponential and polynomial backtracking. All cases of exponential backtracking have been fixed in [#2590](#) and I plan to fix all non-moving polynomial backtracking regexes in [#2597](#).

► Files

★ **RunDevelopment** closed this as completed in [#2584](#) on Oct 25, 2020

yetingli commented on Oct 25, 2020 • edited ▼

Author

Many thanks for your further verification and repair! @RunDevelopment

Before I show the results, one question: Why did some regexes change?
Non-capturing groups were replaced with capturing groups. This is a huge issue because this changed the groups that backreferences were pointing to.

I'm very sorry, these changes have brought you great inconvenience to fix the issues. I want to explain why I have to make these changes (non-capturing -> capturing). Existing static analysis techniques cannot handle non-capturing and backreferences, etc. well. Although the existing dynamic fuzzing methods can process non-capturing and backreferences, etc., the detection is very time-consuming. Meanwhile, for dynamic methods, there will be false negatives.

So I would like to try a combination of dynamic and static methods, that is, some regexes are directly checked by static methods (preferred), and the rest are checked by dynamic fuzzing. I want to try to change the initial regex (e.g., non-capturing -> capturing) so that it can be detected by static methods as much as possible. Thank you for pointing out the problem (This is a huge issue because this changed the groups that backreferences were pointing to), it really cannot be replaced mechanically!

I want to point out that your method seems to favor polynomial runtime over exponential backtracking. Some of the patterns reported as polynomial also have attack strings that can cause exponential backtracking.

This may be related to the detection strategy I mentioned earlier. If I use the dynamic fuzzing method directly, there will be no such problem. It seems that I still need to make a further trade-off between detection efficiency and effect.

I also want to point out that your method has failed to point out a lot of other patterns with exponential backtracking.

On the one hand, there are false negatives as mentioned above, and on the other hand, some characters (e.g., `\xA0-\uFFFF`) or features (e.g., negative lookbehind (`<?!`pattern)) are temporarily not supported by my tool.

I really appreciate your responses :)

There is no doubt that your reply is a great help to the further improvement of my tools. thank you so much again for the terrific work.

RunDevelopment commented on Oct 26, 2020

Member

I'm glad that my results could be of help.

Existing static analysis techniques cannot handle non-capturing and backreferences, etc. well.

I suppose that's the reason flags disappeared as well?

Regarding the non-capturing groups. They can easily be converted to capturing ones if backreferences are also changed so that they still point to the right group.

► Details

Flags are harder to remove.

I don't want to shamelessly promote my own work but you *could* use my library [refa](#).

It can parse a JS regex to get my AST format. With the AST, a modified version of `refa's js.toLiteral` function could create a regex that is equivalent to the original regex but doesn't use any flags. I.e. `/(a|A)+(?=b)/i` will be converted to `/([Aa][Aa])+(?=([Bb]))/`.

The main limitation is that my AST format doesn't support backreferences.


If you were using existing static analyzers that only support ASCII regexes (e.g. RXXR2), then the AST could be transformed to derive an ASCII regex from any JS regex.

That being said, this is probably too much work for too little gain especially if your tool wasn't focused on JS regexes. I just wanted to point out that it is possible.

davisjam commented on Oct 28, 2020 • edited

@yetingli Can you comment on how your technique differs from the work of Rathnayake / Weideman / Wustholz / Shen?

(Might be easier for you to send me an email -- davisjam@purdue.edu)

 inkz mentioned this issue on Nov 23, 2020

[Check] Add more checks for ReDoS [returntocorp/semgrep-rules#747](#)

 Closed

 12 tasks

 rogov-k mentioned this issue on Jun 9

Update prismjs dependency version [jfcere/ngx-markdown#389](#)

 Closed

Assignees

No one assigned

Labels

[language-definitions](#)

Projects


None yet

Milestone

No milestone

Development

Successfully merging a pull request may close this issue.

 Fixed multiple cases of vulnerable REs
[RunDevelopment/prism](#)

3 participants

