Regulator: Dynamic Analysis to Detect ReDoS

Robert McLaughlin, Fabio Pagani, Noah Spahn, Christopher Kruegel, Giovanni Vigna

University of California, Santa Barbara
const normalizeDataURL = (urlString, {stripHash}) => {
  const match = /^data:(?<type>.*),(?<data>.*)(?:#(?<hash>.*))?$/i.exec(urlString);

  if (!match) {
    throw new Error(`Invalid URL: ${urlString}`);
  }

  let {type, data, hash} = match.groups;

  // ...
}
So, About That Code Review ...

```javascript
const normalizeDataURL = (urlString, {stripHash}) => {
  const match = /^data:([^\d\/>\[\]{}\.,;\:\\]+),([^\d\/>\[\]{}\.,;\:\\]+)(?:#([^\d\/>\[\]{}\.,;\:\\]+))?$/i.exec(urlString);

  if (!match) {
    throw new Error(`Invalid URL: ${urlString}`);
  }

  let {type, data, hash} = match.groups;

  // ...
//  data:png,DEADBEEFCAFE
//  data:jpg,DEADBEEFCAFE#value
//  data:,DEADBEEFCAFE
```
const normalizeDataURL = (urlString, {stripHash}) => {
    const match = /^data:(?<type>.*),(?<data>.*)(?:#:(?<hash>.*))?$/i.exec(urlString);

    if (!match) {
        throw new Error(`Invalid URL: ${urlString}`);
    }

    let {type, data, hash} = match.groups;

    // ...
}
const normalizeDataURL = (urlString, {stripHash}) => {
    const match = /^data:(?<type>.*),(?<data>.*)(?:#(?<hash>.*))?$/i.exec(urlString);

    if (!match) {
        throw new Error(`Invalid URL: ${urlString}`);
    }

    let {type, data, hash} = match.groups;

    // ...
}

... but you’re in good company
ReDoS: An Algorithmic Complexity Attack

- **Denial-of-Service (DoS)**
  - Attacker seeks to deny access to a resource
  - Ideally, attacker seeks amplification to achieve asymmetry
  - Consuming *more* victim resources with *less* effort
  - Disrupting system availability, Denying others fair access to victim resources

- **One source of amplification is unintended complexity in victim software**
  - Remote attackers may leverage this to force worst-case execution
  - Consumes execution time on the victim

- **Regular Expressions (regexps) have complexity!**
Significant Real Impact

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 of these websites suffer from at least one ReDoS vulnerability.

_Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers_  
Cristian-Alexandru Staicu and Michael Pradel, USENIX ‘18
Incremental, But Incomplete Prior Results

- Several prior studies examining ReDoS detectors report high false-positive and high true-positive rates on commonly cited tools.
- Shen\(^1\) reports 45% false-negative rate for RXXR2.
- Liu\(^2\) reports 97% false-positive rate for Rexploiter, 48% false-negative rate for Shen’s tool ReScue.
- All have limitations on feature support.

1. Rescue: Crafting Regular Expression DoS Attacks Yuju Shen, Yanyan Jiang, Chang Xu, Ping Yu, Xiaoxing Ma, and Jian Lu, ASE ‘18
2. Revealer: Detecting and Exploiting Regular Expression Denial-of-Service Vulnerabilities Yinxi Liu, Mingxue Zhang, and Wei Meng, Security & Privacy ‘21
Notice: Ambiguity in this NFA means we may need to backtrack!

Attempt constrained depth-first search from q0

When at q2, where do we go?
Overview (automaton) / Prior diagnostic attempts

- Notice: Ambiguity in this NFA means we may need to backtrack!
- Attempt constrained depth-first search from \( q0 \)
- When at \( q2 \), where do we go?

\[
\begin{align*}
\text{abbbbbb}
\end{align*}
\]
Regular Expressions Are Software, Too!

- Much research work has been done on making regular expressions efficient
- BUT this requires long compile times extra space
- Instead, “give up” -- modern regex engines use Spencer’s backtracking alg.
  - Translates a regex into bytecode, as literally as possible

```
/my.?regex(?:goes)here$/
```

```
lexer / parser  ---\   ---\  \-----\  
  \  \         \    \    \  
      \         \    \    \  
compiler     \-----\     \-----\ bytecode
```

```
lexer / parser  ----\   \-----\  
  \  \         \    \    \  
      \         \    \    \  
compiler     \-----\     \-----\ bytecode
```

```
bytecode
```

```
interpreted   
  \        \        
      \        \        
      Match /  no-match
```

```
SomeInput!!
```

```
Interpreter
```

```
Lexer / Parser
```

```
Compiler
```

```
bytecode
```

```
/my.?regex(?:goes)here$/
```
We need a new approach
Regulator: A Simple Solution

/my.+(?:regexp)?here*/

Slowdown Fuzzer -> Formula Derivation -> Dynamic Verifier

Result
Regulator: A Simple Solution

/my.+(?:regexp)?here*/

- Slowdown Fuzzer
- Formula Derivation
- Dynamic Verifier

Result
Regulator / Fuzzer

- Corpus
- Heuristic Prioritization
- Instrumented Execution
- Mutation Engine
Regulator / Fuzzer
Regulator / Fuzzer / Instrumented Execution

- We instrument the regexp bytecode interpreter itself, *Irregexp*
- AFL-style perfmap to efficiently approximate coverage
  - Each cell counts the hits of a particular branch

```
+---+---+---+---+---+
<p>| | | | | |
|   |   |   |   |   |</p>
<table>
<thead>
<tr>
<th>3</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td></td>
<td></td>
<td>53</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td></td>
<td>102</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
+---+---+---+---+---+
```
Problem: We want to prioritize *longest path* while avoiding *local maxima*.

Solution: Prioritize *maximizing representatives* from each perfmap component.

Problem: Only some components are in the *hot path*, prioritizing others is wasted work.

Solution: De-prioritize “stale” inputs with no recent novel offspring inputs.
Regulator / Fuzzer / Mutation Engine “Suggestion”

- The regexp bytecode virtual machine has one *loaded character* register
- We keep a shadow register which remembers *from where in the string* the character was loaded
- On every character-comparison branch, we store
  - (1) the source index of the current loaded character (shadow register)
  - (2) the character(s) which would have negated the branch condition
- Then, during mutation, we will randomly replace the character at that index with one which negates the branch – a “Suggestion”
Regulator: A Simple Solution

Slowdown Fuzzer → Formula Derivation → Dynamic Verifier → Result
Regulator: ReDoS Formula Derivation

- We now have a costly string (the witness)
  - Needs more information! Is this bad? How bad?

Malign inputs take the form: \(\text{attack prefix} + (pump\ string)^n + \text{attack suffix}\)
Regulator: ReDoS Formula Derivation

- Observation: with high likelihood, the *witness string* exemplifies the formula
- Approach: we try all (*prefix*, *pump string*, *suffix*) combinations
- When a we find a slow example, we time 20 sample points from 10 to 256 pumps, and fit the result to either linear, power, or exponential regression

**Regexp:** `hello(1234)*\d*world`

**Witness:** `hello1234worl`  
`prefix`  `pump string`  `suffix`

**Example:**
- `hello1234worl`
- `hello12341234worl`
- `hello123412341234worl`
- ...


Regulator: Dynamic Verification

- Need to verify correctness of derived ReDoS formula
- Approach:
  - Derive the longest attack string under 1 million characters
  - Following prior work, check for 10 seconds of slow-down
- Once verified, we binary-search for the shortest string with 10s slow-down

1. Why aren’t regular expressions a lingua franca? An empirical study on the re-use and portability of regular expressions. James C Davis, Louis G Michael IV, Christy A Coghlan, Francisco Servant, and Dongyoon Lee, ESEC/FSE ‘19
Evaluation

- Against prior work
  - RXXR2
  - Rexploiter
  - NFAA
  - ReScue
  - Revealer
  - PerfFuzz\(^1\) fuzzer + Regulator backend\(^*\)

- Dataset:
  - 3 standard regular expression corpuses:
    - Snort - 13,957 samples
    - RegExLib - 2,990 samples
    - Corpus - 10,037 samples
  - Scrape of NPM (package manager for JavaScript), using both Abstract Syntax Tree crawling and basic constant folding
    - 42,743 samples

Research Question 1: Is Regulator’s Fuzzer Effective?

- We need to ensure high coverage of the regular expression bytecode
- We partition the samples into thirds by bytecode instruction count
Research Question 2: How Does Regulator Compare?

Other ReDoS detection tools (*base dataset*)

<table>
<thead>
<tr>
<th>Tool</th>
<th>Corpus</th>
<th></th>
<th>RegExLib</th>
<th></th>
<th>Snort</th>
<th></th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sup.</td>
<td>TP</td>
<td>FP</td>
<td>FN</td>
<td>Sup.</td>
<td>TP</td>
<td>FP</td>
</tr>
<tr>
<td>RXXR2</td>
<td>11,696</td>
<td>30</td>
<td>26</td>
<td>2,154</td>
<td>2,301</td>
<td>100</td>
<td>27</td>
</tr>
<tr>
<td>Rexploiter</td>
<td>10,536</td>
<td>293</td>
<td>973</td>
<td>1,891</td>
<td>1,764</td>
<td>98</td>
<td>287</td>
</tr>
<tr>
<td>NFAA</td>
<td>11,256</td>
<td>738</td>
<td>952</td>
<td>1,446</td>
<td>1,977</td>
<td>279</td>
<td>70</td>
</tr>
<tr>
<td>ReScue</td>
<td>12,441</td>
<td>26</td>
<td>4</td>
<td>2,158</td>
<td>2,780</td>
<td>115</td>
<td>14</td>
</tr>
<tr>
<td>Revealer</td>
<td>13,206</td>
<td>428</td>
<td>30</td>
<td>1,756</td>
<td>2,946</td>
<td>245</td>
<td>17</td>
</tr>
<tr>
<td>REGULATOR</td>
<td>13,595</td>
<td>2,156</td>
<td>0</td>
<td>28</td>
<td>2,973</td>
<td>519</td>
<td>0</td>
</tr>
</tbody>
</table>

Total: 26,605 4,847 0 99
Research Question 2: How Does Regulator Compare?

Other ReDoS detection tools (npm dataset)

<table>
<thead>
<tr>
<th>Tool</th>
<th>NPM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sup.</td>
</tr>
<tr>
<td>RXXR2</td>
<td>35,842</td>
</tr>
<tr>
<td>Rexplorer</td>
<td>30,317</td>
</tr>
<tr>
<td>NFAA</td>
<td>32,158</td>
</tr>
<tr>
<td>ReScue</td>
<td>39,258</td>
</tr>
<tr>
<td>Revealer</td>
<td>38,379</td>
</tr>
<tr>
<td>REGULATOR</td>
<td>41,342</td>
</tr>
</tbody>
</table>
Research Question 3: Novel Detections?

- 6 assigned CVE numbers
- Dozens of verified & fixed bugs
- Several difficult-to-find detections
  - Requires certain flags to be set
  - High complexity trips up other analyses
  - Feature use: back-references, look-arounds, special character groups, etc
A Recommendation

- Limit attacker-controlled string lengths
- A limit of 1,000 or 10,000 characters mitigates nearly all ReDoS that we observed.
Thank you!

https://github.com/ucsb-seclab/regulator-dynamic

@robmcl4

robert349@ucsb.edu

/my.+(?:regexp)?here*/