Research Study of the Regular Expression Denial of Service (ReDoS) security vulnerability.

Anonymous Participant ID: [ ]

Dear participant,

Thank you for participating in our study. We will ask you to do a couple of exercises related to the detection and repair of regular expressions that may be vulnerable to the ReDoS security vulnerability. One of our studied regexes comes from your contributions to a public source code repository.

Our study is composed of 2 exercises, and will require a total of about 1 hour of your time. We will record the audio of our conversation.

We hope that this study helps you improve the security of the software that you write.

0. Training: Information about the ReDoS Security Vulnerability

How ReDoS Attacks Work.

At a high level, ReDoS attacks use an expensive regex query to overload the CPU of a server-side process, reducing the number of connections that it can service. ReDoS attacks have two requirements. **First**, the regex engine used by the victim must use **backtracking** to evaluate inputs against regexes. Unfortunately, this requirement is met by most modern and popular programming languages — e.g., JavaScript-V8 (Node.js), Python, Java, C++-11, C#-Mono, PHP, Perl, and Ruby. **Second**, the victim must evaluate input provided by the attacker against a **vulnerable regex**. If both conditions are met, the attacker carries out a ReDoS exploit by sending **malign input** to the victim. A malign input forces the regex engine to explore a vast search space to check whether the input matches the regex — which correspondingly causes the CPU overload. Because each step in the search requires backtracking, this phenomenon is known as **catastrophic backtracking**.

Catastrophic Backtracking.

At the core of most regex engines is a backtracking-based search algorithm. Regex engines accept a **regex** describing a language, and an **input** to be tested for membership in this language. A backtracking-based regex engine can be thought of as constructing a non-deterministic finite automaton (NFA) from the regex and then simulating the NFA on the input. To simulate non-determinism, whenever the engine makes a choice, it pushes the current NFA state onto a stack of backtracking points. If a mismatch occurs, the engine backtracks recursively, trying alternative decisions until it either finds a match or exhausts the set of backtracking points. Catastrophic backtracking is the term for the result when a regex engine has to explore an enormous number of states before it can declare a match or mismatch for an input. Typically the number of explored states is polynomial (commonly $O(n^2)$ or $O(n^3)$) or exponential in the input length.

Vulnerable Regexes and Malign Inputs.

A regex is vulnerable to catastrophic backtracking when its evaluation may require the regex engine simulating the NFA to make a large amount of non-deterministic choices on malign input. A malign input has three components: a *prefix* string, a *pump* string, and a *suffix* string. One constructs malign input by concatenating the components as follows: *malign input* = *prefix* + *pump (repeated 1 or more times)* + *suffix*. The more times a malign input repeats the pump string, the more potent it is.

Here is the effect of the formula for malign input. The prefix brings the NFA to a set of ambiguous states. These states are called ambiguous because on a special subsequent input (the pump), the NFA can move from one of these states to another by more than one path. If it sees such an input, a backtracking regex engine will try one path and save backtracking points for the other(s). By repeating the pump, a malign input expands the search space by forcing repeated choices, building up the stack of backtracking states. A final suffix causes a mismatch, triggering polynomial or exponential backtracking through the stack of saved states.

A real-world example: ReDoS in Python.

We illustrate ReDoS by describing a vulnerable regex found in the Python core library *difflib*. This regex is represented below. Its ambiguous states are the two “whitespace” nodes. The components of the malign input are: an empty prefix, a pump of “any whitespace character”, and a suffix of “any non-whitespace character”.

Example of vulnerable regex `/\s*#?\s*$/` and its corresponding NFA diagram.

Let's see what happens on a malign input. The (empty) prefix brings us to the first ambiguous whitespace node. When the regex engine encounters each pump, it must make a choice: to stay in the first whitespace node, or to advance to the second by skipping the optional '#' node. The regex engine will choose first to advance when possible. When the suffix causes a mismatch, the regex engine revisits all of its choices and tries the other one, which was to stay in the first node. Because this happens for every whitespace character in the string, the search space is equivalent to an $O(n^2)$ doubly-nested traversal of the string.

Since this regex could be used by a Python-based server to process user input, it represents a ReDoS vulnerability. For this regex, on a typical workstation, multi-second slowdowns will begin to manifest when the malign input is around 1,000 characters long. Vulnerable regexes with $O(n^2)$ complexity evaluate quickly on normal input and also on short malign inputs. Meanwhile, vulnerable regexes with exponential complexity will exhibit slow match times even on short malign inputs, e.g., 30 characters long.

1. Exercise 1. Detecting ReDoS

The purpose of this exercise is to compare two approaches for detecting ReDoS: (1) an automatic tool, and (2) a set of anti-patterns. We will ask you to use them and tell us your impressions about them.

1.1. Detecting ReDoS using only automatic detection tools.

1.1.1. Training:

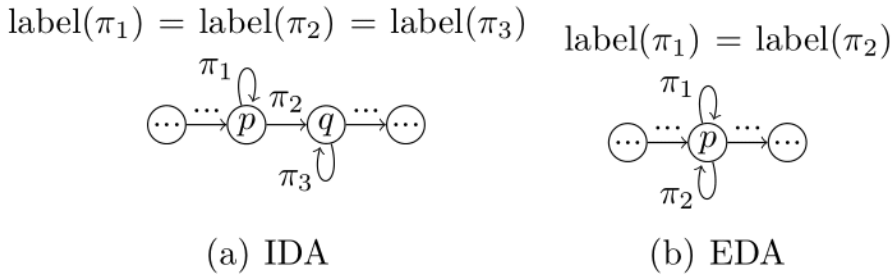
This exercise uses an automatic tool that can detect whether a regex is vulnerable to ReDoS, performing static analysis on it, checking whether catastrophic backtracking could occur. The tool detects two types of catastrophic backtracking, namely polynomial backtracking and exponential backtracking. The tool inspects the underlying NFA of a regex, looking for the ambiguity mentioned earlier in Section "0. Training". The tool identifies regexes with a Polynomial Degree of Ambiguity (IDA), an Exponential Degree of Ambiguity (EDA), or no ambiguity. Regexes with IDA are vulnerable to polynomial backtracking, and with EDA, exponential backtracking.

The following figure illustrates the patterns for IDA and EDA below. Here is a guide for the notation:

- The NFAs are drawn using nodes and edges
- Some edges are labeled with a Greek "pi" symbol, like "π_1". This indicates a problematic sub-pattern within the regex. The pi symbol is the shorthand for a pattern.

- The notation “label(π_1)” represents a string matched by the pattern π_1. For example, label(a+b) could be ab or aab. Thus, when it says “label(π_1) = label(π_2)”, it means that the two sub-patterns π_1 and π_2 can both match some string, implying the possibility of ambiguity during the regex match.

Illustration of Polynomial (IDA) and Exponential (EDA) Degree of Ambiguity in the NFA:



1.1.2. Exercise:

Please, consider the regular expressions below. For each one of them, we also show the output of the automatic tool for detecting ReDoS. Please, inspect each regular expression and its corresponding tool output, and answer the questions beside them. Please, feel free to also use the reference material below:

Explanation of regex (and Syntax): https://regex101.com/
Regex NFA visualization: https://regexper.com/
Automatic tool for detecting ReDoS: https://www.regextools.io/analyze

Regular expression	**Output of automatic detection tool**	**How strongly do you understand what makes this regex vulnerable, given the explanation provided by the tool?**	**Can you explain your reasoning?**
`([a-z]+)0+(\d+)`	Contains IDA, degree 1, with: a0000...0001a IDA exploit string as JSON: {"degree":1,"separators":["a00"],"pumps":["00"],"suffix":"01a"} Prefix: "a00" Pump 0: "00" Suffix: "01a"	[ ] Very strongly understand [ ] Strongly understand [ ] Neutral understanding [ ] Weakly understand [ ] Very weakly understand [ ] It is not vulnerable	
`(?:coast)[\s](\d+(\.?_?\d+)+)`	Contains EDA with: coast\x0900000...00c	[ ] Very strongly understand [ ] Strongly understand	

	EDA exploit string as JSON: {"degree":0,"separators":["coast\t000"],"pumps":["00"],"suffix":"c"} Prefix: "coast\x09000" Pump: "00" Suffix: "c"	[ ] Neutral understanding [ ] Weakly understand [ ] Very weakly understand [ ] It is not vulnerable	
`AppleWebKit\/([0-9]+)(?:\.([0-9]+))(?:\.([0-9]+))`	EDA analysis performed in: 13ms Does not contain EDA IDA analysis performed in: 17ms Does not contain IDA	[ ] Very strongly understand [ ] Strongly understand [ ] Neutral understanding [ ] Weakly understand [ ] Very weakly understand [ ] It is not vulnerable	

1.2. Detecting ReDoS using automatic detection tools and anti-patterns.

1.2.1. Training:

Now, please read and understand the anti-patterns below. They describe regular expressions that can be vulnerable to ReDoS in terms of the regex pattern language, rather than an NFA. Note: we use the * symbol to denote any unbounded quantifier, such as *, +, or {1,}

Name	**Description**	**Example**
Concat 1	R = ...P*Q*... (R has a sub-regex P*Q*): The two quantified parts P* and Q* can match some shared string s.	a*(aa)*, both can match aa.
Concat 2	R = ...P*SQ*... (R has a sub-regex P*SQ*): The two quantified parts P* and Q* can match a string s from the middle part S.	(a|b)+ab(ab)+, quantified parts (a|b)+ and (ab)+ can match the middle part ab.
Concat 3	R = ...P*S*Q*... (R has a sub-regex P*S*Q*): Advanced form of Concat 1. Since S* includes an empty string, the ambiguity between P* and Q* can be realized.	a*b*a*, b* can be skipped, so both a* can match aa. a*b?a*, b? can be skipped, so both a* can match aa.
Star 1	R*, R= (P|Q|...): There is an intersection between any of the two alternates i.e., both can match some shared string s.	(\w|\d)+, both \w and \d can match digits [0-9].

Star 2	R*, R= (P|Q|...): You can make one option of the alternation by repeating another option multiple times or by concatenating two or more options multiple times.	(x|y|xy)*, we can create the 3rd option xy by adding the first option x and second option y.
Star 3	R*, R= (...P*...): Nested quantifiers, but only if RR would follow any of the concat antipatterns above.	(a*)*, we check a*a* and find that it is infinitely ambiguous by Concat 1. (xy*)* is not Star 3. We check xy*xy* and find that it is not infinitely ambiguous by any Concat.

1.2.2. Exercise:

After you have understood the anti-patterns above, please reassess the same regular expressions, according to your new understanding. Please, feel free to also use the reference material below:

Automatic tool for detecting ReDoS: https://www.regextools.io/analyze
Explanation of regex (and Syntax): https://regex101.com/
Regex NFA visualization: https://regexper.com/

Regular expression	**Output of detection tool**	**Write here the anti-pattern that the regex fits (if it does)**	**How strongly do you understand what makes this regex vulnerable, given the explanation provided by the tool and the anti-patterns?**	**Can you explain your reasoning?**
`([a-z]+)0+(\d+)`	Contains IDA, degree 1, with: a0000...0001a IDA exploit string as JSON: {"degree":1,"separators":["a00"],"pumps":["00"],"suffix":"01a"} Prefix: "a00" Pump 0: "00" Suffix: "01a"		[ ] Very strongly understand [ ] Strongly understand [ ] Neutral understanding [ ] Weakly understand [ ] Very weakly understand [ ] It is not vulnerable	
`(?:coast)[\s](\d+(\.?_?\d+)+)`	Contains EDA with: coast\x0900000...00c EDA exploit string as JSON: {"degree":0,"separators":["coast\t00"],"pumps":["00"],"suffix":"c"} Prefix: "coast\x09000" Pump: "00"		[ ] Very strongly understand [ ] Strongly understand [ ] Neutral understanding [ ] Weakly understand [ ] Very weakly understand [ ] It is not vulnerable	

	Suffix: "c"			
`AppleWebKit\/([0-9]+)(?:\.([0-9]+))(?:\.([0-9]+))`	EDA analysis performed in: 13ms Does not contain EDA IDA analysis performed in: 17ms Does not contain IDA		[ ] Very strongly understand [ ] Strongly understand [ ] Neutral understanding [ ] Weakly understand [ ] Very weakly understand [ ] It is not vulnerable	

1.3. Finally:

1. In general terms, after having learned the anti-patterns, how helpful do you think they will be for you in your future efforts to detect ReDoS?

[ ] Very helpful
[ ] Helpful
[ ] Neutral
[ ] A little helpful
[ ] Not helpful

2. Can you explain your reasoning??

3. In general terms, which strategy do you think would be more helpful to you in your future efforts to detect ReDoS?

[ ] Applying the detection tool only
[ ] Applying the anti-patterns only
[ ] Applying the detection tool and the anti-patterns combined together
[ ] Applying the detection tool in some cases and the anti-patterns in other cases

4. Can you explain your reasoning?

5. Are there any specific contexts in which you would prefer using only the tool, or only the anti-patterns for detecting vulnerable regexes? Please, explain.

2. Exercise 2. Repairing ReDoS (20 minutes).

In this second exercise, we will compare two approaches for repairing the ReDoS vulnerability of a regular expression: (1) an automatic repair tool, and (2) a set of anti-patterns and fixes for them. We will ask you to use them and tell us your impressions.

2.1. Repairing ReDoS using automatic repair tools.

2.1.1. Training:

This exercise uses an automatic tool that can repair a regular expression that is vulnerable to ReDoS. The result of the tool is a regex that is guaranteed to be semantically equivalent to the original, but that is no longer vulnerable to ReDoS. Here are some details about how the tool works. The tool converts a regex's non-deterministic finite automaton (NFA) into an equivalent deterministic finite automaton (DFA). That DFA is unambiguous, by definition – on every possible input, there is always a single (deterministic) choice. The tool then converts this DFA back into an equivalent regex using an algorithm that ensures a correspondence between paths in the original DFA and paths in the constructed regex.

2.1.2. Exercise:

We found the following vulnerable regular expression in your code base. It might expose the users of your project to ReDoS.

Software project: https://github.com/stevelittlefish/littlefish
File: littlefish/validation.py
Introduced on: March 19, 2016, Line 13.
Regular expression:

```
telephone_regex = re.compile('^\+?\d+(\s*-?\s*\d+)*$')
```

Used in:

```
def validate_telephone_number(telephone_number):
    return telephone_regex.match(telephone_number) and
len(telephone_number) >= 8
```

How strongly would you say that you understand this regular expression?

- [ ] Very strongly understand
- [ ] Strongly understand
- [ ] Neutral understanding
- [ ] Weakly understand
- [ ] Very weakly understand

We executed the automatic repair tool to fix this regular expression, to remove its vulnerability to ReDoS. We include the output of this tool below.

Please answer the following questions about the solution proposed by the automatic ReDoS repair tool. Please, feel free to also use the reference material below:

Automatic tool for detecting ReDoS: https://www.regextools.io/analyze
Automatic tool for repairing ReDoS: https://www.regextools.io/rewrite
Explanation of regex (and Syntax): https://regex101.com/
Regex NFA visualization: https://regexper.com/

Your vulnerable regular expression: `^\+?\d+(\s*-?\s*\d+)*`

Output of the automatic ReDoS repair tool:

```
((\+)((\d(\s\s*\d|\d)*(\s\s*\-|\-))(\d(\s\s*\d|\d)*(\s\s*\-|\-)|\s)*(\d(\s\s*\d|\d)*)|\d(\s\s*\d|\d)*)|(\d(\s\s*\d|\d)*(\s\s*\-|\-))(\d(\s\s*\d|\d)*(\s\s*\-|\-)|\s)*(\d(\s\s*\d|\d)*)|\d(\s\s*\d|\d)*)
```

How strongly would you say that you understand the regular expression repaired by the automatic tool?

[ ] Very strongly understand
[ ] Strongly understand
[ ] Neutral understanding
[ ] Weakly understand
[ ] Very weakly understand

Can you explain your reasoning?

Generally, how strongly would you say that you understand what makes the regular expression repaired by the automatic tool not vulnerable to ReDoS any more?

[ ] Very strongly understand
[ ] Strongly understand
[ ] Neutral understanding
[ ] Weakly understand
[ ] Very weakly understand

Can you explain your reasoning?

How comfortable would you be with replacing the vulnerable regular expression with the one repaired by the automatic tool in your code base?

[ ] Very comfortable

[ ] Comfortable
[ ] Neutral
[ ] Uncomfortable
[ ] Very uncomfortable

Can you explain your reasoning?

2.2. Repairing ReDoS using automatic repair tools and anti-patterns.

2.2.1. Training:

Please, consider again the anti-patterns that describe regular expressions that can be vulnerable to ReDoS. This time, we also show examples of how other developers repaired regular expressions under each anti-pattern. Please, review the anti-patterns again and the example fixes. Note: we use the * symbol to denote any not-upper-bounded quantifier: *, +, or {1,}

Name	**Description**	**Anti-pattern Example**	**Some Fix strategies**	**Fix Example**
Concat 1	R = ...P*Q*... (R has a sub-regex P*Q*): The two quantified parts P* and Q* can match some shared string s.	a*(aa)*, both can match aa.	F1: Add a delimiter S between P* and Q* F2: Reduce P and/or Q F3: Merging P and Q to form S F4: Reduce or remove repetitions on P and/or Q F5: Remove or substantially modify P and Q	F1: For a*(aa)*, add ‘b’ between the quantifiers i.e., a*b(aa)* F2: For \w*\d*, change \w to [a-zA-Z_] i.e., [a-zA-Z_]*\d+ F3: For \d*\w*, merge \w and \d i.e., \w* F4: For \w*\d*, change the regex to \w\d* or \w*\d or \w\d For \w*\d*, change the regex to \w{1,30}\d{1,30} F5: For \d*\w*\d*, change the regex to \w+ and add logic to catch digits at the start and at the end
Concat 2	R = ...P*SQ*... (R	(a|b)+ab(ab)+,	F1: Add a delimiter T between P* and	F1: For (a|b)*ab(ab)*, add ‘c’ between

	has a sub-regex P*SQ*): The two quantified parts P* and Q* can match a string s from the middle part S.	quantified parts (a|b)+ and (ab)+ can match the middle part ab.	S, or between S and Q* F2: Reduce P and/or Q F4: Reduce or remove repetitions on P and/or Q F5: Remove or substantially modify P and Q	(a|b)* and ab i.e., (a|b)*cab(ab)* F2: For, \S+@\S+, change the first \S i.e., [^@\s]+@\S+ F4: For (a|b)*ab(ab)*, change it to (a|b)ab(ab)* or (a|b)*ab(ab) or (a|b)ab(ab) For (a|b)*ab(ab)*, change it to (a|b)ab(ab){1,30} or (a|b){1,30}ab(ab){1,30} F5: For (a|b)*ab(ab)*, change the regex to (ab)+
Concat 3	R = ...P*S*Q*... (R has a sub-regex P*S*Q*): Advanced form of Concat 1. Since S* includes an empty string, the ambiguity between P* and Q* can be realized.	a*b*a*, b* can be skipped, so both a* can match aa. a*b?a*, b? can be skipped, so both a* can match aa.	F1: Make S not optional (e.g., S+) F2: Reduce P and/or Q F4: Reduce or remove repetitions on P and/or Q F5: Remove or substantially modify P and Q	F1: For a*b*a*, change it to a*b+a* or change the regex to a*b+a*|a* F2: For \w+:?\d+, change \w to [a-zA-Z_] i.e., [a-zA-Z_]+:?\d+ F4: For \w+:?\d+, change it to \w:?\d+ or \w+:?\d or \w:?\d For \w+:?\d+, change it to \w{1,30}:?\d{1,30} F5: For \w+:?\d+, change the regex to \d+ and add logic to match letters at the start
Star 1	R*, R= (P|Q|...): There is an intersection between any of the two alternates i.e., both can match some shared string s.	(\w|\d)+, both \w and \d can match digits [0-9].	F1: Add a delimiter S to P or Q (PS|Q|...)*, (SP|Q|...)*, (P|SQ|...)*, (P|QS|...)* F2: Reduce P and/or Q F3: Reducing P and Q so that they longer generate shared string and add shared string in disjunction F4: Reduce or remove repetitions on P and/or Q F5: Remove or substantially modify P and Q	F1: For (\w|\d)+, Change (\w|\d)+ to (\w:|\d)+ or (:\w|\d)+ or (\w|:\d)+ or (\w|\d:)+ F2: For (\w|\d)+, change \w to [a-zA-Z_] i.e., ([a-zA-Z_]|\d)+ F3: For (\w|\d)+, change the regex to ([a-zA-Z_]|\d)+ F4: For (\w|\d)+, change the regex to (\w|\d) For (\w|\d)+, change the regex to (\w|\d){1,30}

				F5: (\w|\d)+, change the regex to (\w+|\d+)
Star 2	R*, R= (P|Q|...): You can make one option of the alternation by repeating another option multiple times or by concatenating two or more options multiple times.	(x|y|xy)*, we can create the 3rd option xy by adding the first option x and second option y.	F1: Add a delimiter S to P or Q (PS|Q|...)*, (SP|Q|...)*, (P|SQ|...)*, (P|QS|...)* F2: Reduce P and/or Q F3: Reducing P and Q so that they longer generate shared string and add shared string in disjunction F4: Reduce or remove repetitions on P and/or Q F5: Remove or substantially modify P and Q	F1: For (a|b|ab)+, change the regex to (ac|b|ab)+ or (a|bc|ab)+ or (a|b|abc)+ F2: For (ab|cd|abcd)+, change the regex to (a|cd|abcd)+ F3: For (a|b|ab)+, change the regex to (a|b)+ (it can generate any number of a, b or ab) F4: For (a|b|ab)+, change the regex to (a|b|ab) For (a|b|ab)+, change the regex to (a|b|ab){1,30} F5: For (a|b|ab)+, change the regex to a+|b+|(ab)+
Star 3	R*, R= (...P*...): Nested quantifiers, but only if RR would follow any of the concat antipatterns above.	(a*)*, we check a*a* and find that it is infinitely ambiguous by Concat 1. (xy*)* is not Star 3. We check xy*xy* and find that it is not infinitely ambiguous by any Concat.	F1: adding a delimiter S to R i.e., SR* or R*S so that S and R can not generate any shared string. F2: Reduce P. F4: Removing outer quantifier or Limiting repetitions on both outer and inner quantifiers. Note: too low repetition bounds may limit the language too much, and too high repetition bounds may still contain too many states. F5: Remove or substantially modify the regex	F1: For (:?a*:?)+, change the regex to (:a*:?)+ or (:?a*:)+ F2: For (0\d+)+, change the regex to (0[1-9])+ F4: For (a*)*, change the regex to a* For (:?a*:?)+, change the regex to (:?a{0,20}:?){1,30} F5: For (:?a*:?)+, change the regex to a* and add logic to match optional ':' at the start and at the end

2.2.2. Exercise:

Now, please consider your vulnerable regular expression again. We would like to ask you to fix this regular expression yourself, to make it not vulnerable to ReDoS. To help with this task, feel free to consider the output of: the automatic repair tool, the anti-patterns, and the fix strategies

described above. We ask that you make your own judgment and fix the regular expression in whatever way you feel is appropriate. Please, feel free to also use the reference material below:

Automatic tool for detecting ReDoS: https://www.regextools.io/analyze
Automatic tool for repairing ReDoS: https://www.regextools.io/repair
Explanation of regex (and Syntax): https://regex101.com/
Regex NFA visualization: https://regexper.com/

Your vulnerable regular expression: `^\+?\d+(\s*-?\s*\d+)*`

Output of the automatic ReDoS repair tool :

```
((\+)((\d(\s\s*\d|\d)*(\s\s*\-|\-))(\d(\s\s*\d|\d)*(\s\s*\-|\-)|\s)*(\d(\s\s*\d|\d)*)|\d(\s\s*\d|\d)*)|(\d(\s\s*\d|\d)*(\s\s*\-|\-))(\d(\s\s*\d|\d)*(\s\s*\-|\-)|\s)*(\d(\s\s*\d|\d)*)|\d(\s\s*\d|\d)*)
```

Please, write below your own repair for this regular expression:

Generally, how strongly would you say that you understand your repaired regular expression?

[ ] Very strongly understand
[ ] Strongly understand
[ ] Neutral understanding
[ ] Weakly understand
[ ] Very weakly understand

Can you explain your reasoning?

Generally, how strongly would you say that you understand what makes your repaired regular expression not vulnerable to ReDoS any more?

[ ] Very strongly understand
[ ] Strongly understand
[ ] Neutral understanding
[ ] Weakly understand
[ ] Very weakly understand

Can you explain your reasoning?

How comfortable would you be with replacing the vulnerable regular expression with the one that you repaired in your code base?

[ ] Very comfortable

[ ] Comfortable
[ ] Neutral
[ ] Uncomfortable
[ ] Very uncomfortable

Can you explain your reasoning?

2.3. Finally:

1. In general terms, after having learned the anti-patterns, how helpful do you think they will be for you in your future efforts to repair ReDoS?
   [ ] Very helpful
   [ ] Helpful
   [ ] Neutral
   [ ] A little helpful
   [ ] Not helpful

2. Can you explain your reasoning?

3. In general terms, which strategy do you think would be more helpful to you in your future efforts to repair ReDoS?
   [ ] Applying the repair tool only
   [ ] Applying the anti-pattern fixes only
   [ ] Applying the repair tool and the anti-pattern fixes combined together
   [ ] Applying the repair tool in some cases and the anti-pattern fixes in other cases

4. Can you explain your reasoning?

5. Are there any specific contexts in which you would prefer only the tool, or only the anti-patterns for repairing a vulnerable regex? Please, explain.

6. In this exercise, you have used some tools. Please mark the ones you used and tell us how they were useful?
   [ ] regex101.com
   Note:

[ ] https://regexper.com/
Note:

[ ] https://www.regextools.io/analyze
Note:

[ ] https://www.regextools.io/rewrite
Note:

3. Demographic Questions (5 minutes)

Q1. How long have you worked as a professional software developer? (Internships/co-ops also count)

[ ] No Experience
[ ] Less than one year
[ ] 1-2 years
[ ] 3-5 years
[ ] 6-10 years
[ ] more than 10 years

Q2. Estimate your regular expression expertise level:

Check the box that best describes your expertise	Features
[ ] Beginner Knows any of the features that appear in the right column.	one-or-more repetition. Example: z+ a capture group. Example: (caught) zero-or-more repetition. Example: .* custom character class. Example: [aeiou] Custom class with a range. Example: [a-z] start-of-line. Example: ^ end-of-line. Example: $ Shortcut for any non-newline char. Example: .
[ ] Intermediate Knows any Beginner features and any of the features that appear in the right column.	negated custom character class. Example: [^qwxf] logical OR. Example: a|b - zero-or-one repetition. Example: z? Non-greedy repetition (as few as possible). Example: z+? Shortcut for [\t \n \r \v \f 'space']. Example: \s Shortcut for [0123456789]. Example: \d Shortcut for [a-zA-Z0-9 _]. Example: \w
[ ] Expert	group without capturing. Example: a(?:b)c

Knows any Beginner features, any Intermediate features and any of the features that appear in the right column.	named capture group. Example: (?P<name>x) exactly n repetitions. Example: z{8} at least n repetitions. Example: z{15,} n ≤ x ≤ m repetitions. Example: z{3,8} sequence doesn't follow. Example: a(?!yz) word/non-word boundary. Example: \b negated boundary. Example: \B Lookahead–matching sequence follows. Example: a(?=bc) options wrapper. Example: (?i)CasE sequence doesn't precede. Example: (?<!x)yz Lookbehind–matching sequence precedes. Example: (?<=a)bc absolute end of string. Example: \Z match the ith capture group. Example: \1 reference named capture group. Example: (?P=name) match U+000B. Example: \v Shortcut for non-word. Example: \W Shortcut for non-decimal. Example: \D Shortcut for non-whitespace. Example: \S

Q3. Did you know about ReDoS before participating in this study?

[ ] Yes
[ ] No

Q4. Have you fixed any regex vulnerable to ReDoS before?

[ ] Yes
[ ] No

Q5. In general, to reflect about regex behavior, do you prefer to think about their NFA or about the regex itself?

[ ] The NFA
[ ] The regex itself
[ ] Both, most of the time
[ ] Sometimes the NFA, sometimes the regex itself?

Q6. Can you explain your reasoning?