PLAY HAPPY SOUND FOR PLAYER TO ENJOY

UNDERSTANDING CATASTROPHIC REGEX BACKTRACKING

INTRO

WHAT IS REGEX

RegEx stands for ‘Regular Expressions’. It represents a search pattern used by string-searching algorithms for find, find and replace, or input validation.

HAPPY SOUND

2

Regex is essentially a pattern that search engines use to match strings, enabling for find, find and replace activities or input validation. Regex originated back in 1951 blah blah history, but because heavily used in the late 60s for pattern matching and lexical analysis.

Who knows What GREP stands for?
Globally search for a Regular Expression and Print matching lines.

How many people know regex? Of those, who has had to learn regex the same amount of times they’ve had to use it.

Often people say that you started with a problem and decided to use regex, now you have two problems. This talk will show just how true that is.

\s	whitespace
\S	not whitespace
\d	digit
\D	not digit
\w	word
\W	not word

CHARACTER CLASSES

*	0 or more
+	1 or more
?	0 or 1
{3}	exactly 3
{3,}	3 or more
{3,5}	3 to 5

QUANTIFIERS

.	any character
(a|b)	a or b
(…)	group
[abc]	range (a or b or c)
[^abc]	not (a or b or c)
[a-q]	range (a to q)

GROUPS AND RANGES

CHEAT SHEET

HAPPY SOUND 3

Regex can be roughly broken up into three types of operators.

This list isn't exhaustive, but will provide necessary context for the following slides.

Lets take an example of a simple pattern we want to match – an IP Address. Could be many different reasons.

```
\d+\.\d+\.\d+\.\d+
```

IP ADDRESS

\d+\.\d+\.\d+\.\d+

192.168.0.1

IP ADDRESS

\d+\.\d+\.\d+\.\d+

192.168.0.2222222222222

IP ADDRESS

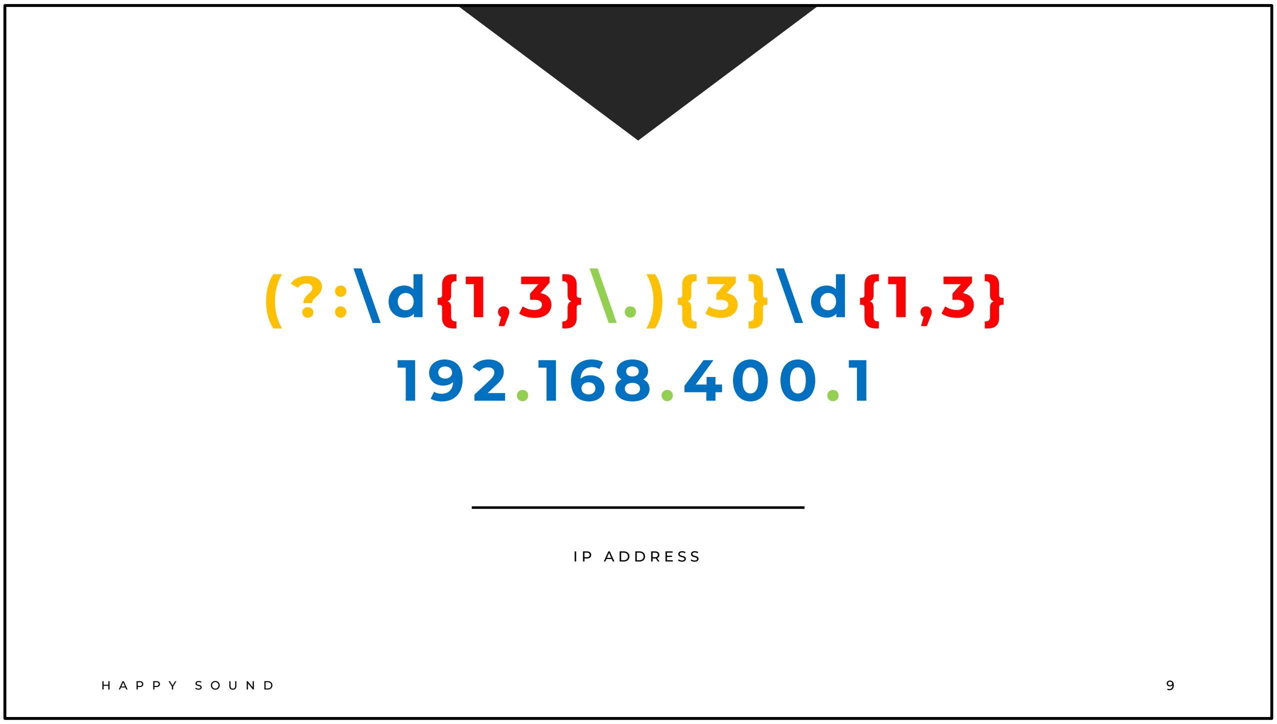
HAPPY SOUND

7

Unfortunately we can see the pattern doesn't properly validate what we're looking for.

```
(?:\d{1,3}\.){3}\d{1,3}
```

IP ADDRESS



Still it leaves room for improvement

(?:1?(?:1?\d{1,2}|2[0-4]\d|25[0-5])\.){3}(?:1?\d{1,2}|2[0-4]\d|25[0-5])

IP ADDRESS

2001:DB8:85A3:8D3:1319:8A2E:370:7348

IPV6 ADDRESS

```
((?:(?:1?(1?\d{1,2}|2[0-4]\d|25[0-
5])\.(1??\d{1,2}|2[0-4]\d|25[0-5])\.(1?\d{1,2}|2[0-
4]\d|25[0-5])\.(1?\d{1,2}|2[0-4]\d|25[0-
5])))|(?:(?:1?((?:(?:[0-9A-Fa-f]{1,4}:){7}[0-9A-Fa-
f]{1,4}|(?=(?:[0-9A-Fa-f]{1,4}:){0,7}[0-9A-Fa-
f]{1,4}\z)|(([0-9A-Fa-f]{1,4}:){1,7}|:)((:[0-9A-Fa-
f]{1,4}){1,7}|:))))))
```

SMILEYFACE GUNEMOJI

BACKTRACKING

A\d+.	A	1	2	3	4	
A\d+.	A	1	2	3	4	

Here's another example of how backtracking works even though there 'appears' to be a match.

Progressing through the first line the regex fails because the . can't match anything.

The regex backtracks to A123, matches 0 non-digits, then matches 4 as the .

BACKTRACKING

/\d-\d+$/g	0	4	0	8	-	0	1	7	-	8	4	2
/\d-\d+$/g	0	4	0	8	-	0	1	7	-	8	4	2
/\d-\d+$/g	0	4	0	8	-	0	1	7	-	8	4	2
/\d-\d+$/g	0	4	0	8	-	0	1	7	-	8	4	2
/\d-\d+$/g	0	4	0	8	-	0	1	7	-	8	4	2
/\d-\d+$/g	0	4	0	8	-	0	1	7	-	8	4	2
/\d-\d+$/g	0	4	0	8	-	0	1	7	-	8	4	2

Very simplified vision of how backtracking works. What we're looking for is the last four digits of a phone number – Verification questions?

As the regex progresses through the phone number it has to backtrack several times as the match fails.

STACK OVERFLOW OUTAGE

JULY 20, 2016

1444 User nguyentrugkien submits a question: *"Join tiles in Corona SDK into one word for a Breakout game grid?"*

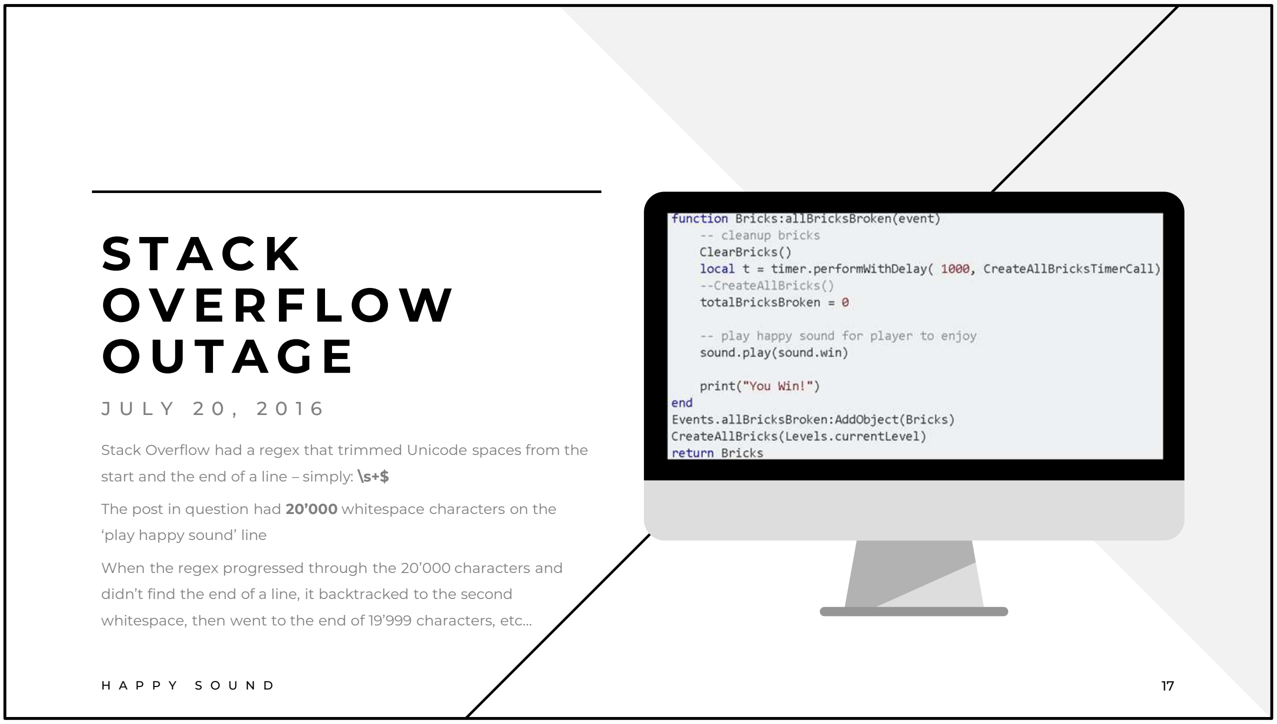
1454 Memory dump leads Stack Overflow team to the specific regex and the 'Front Page' (last 3000 posts). Reran all posts against the regex to find which post took the longest to run.

1508 Line trimming rewritten to be substring function.

1518 Fix rolled out and Stack Overflow becomes available again.

STACK OVERFLOW OUTAGE

JULY 20, 2016

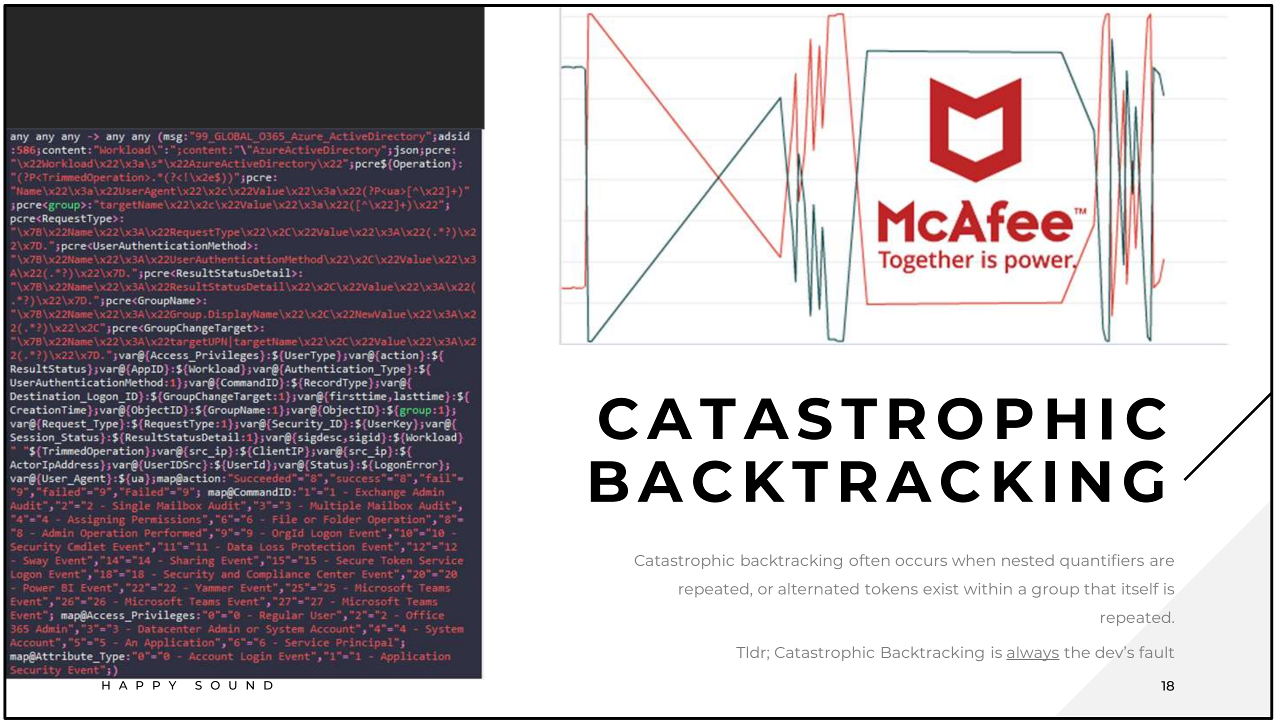


So what caused the stackoverflow outage?

They had a regex element that trimmed Unicode spaces from the start or end of a line. To break a long regex down to the simple element that caused the problem \s$

Somehow, when the user copied part of their code to stackoverflow, they managed to include 20’000 whitespaces on the line “– play happy sound for player to enjoy”

The regex would start at the beginning, progress through all 20’000 characters, then fail as the line didn’t end.
It would backtrack, and start at the second character, for all 19’999 characters, then fail as the line didn’t end.
It would backtrack and start at the third character, for all 19’998 characters, then …



This is what's known as catastrophic backtracking. Why do I know so much about this? Had to deal with an outage myself courtesy of McAfee.

On the left-hand side here we can see an example of a parsing file that takes O365 Azure AD events for their SIEM product. Unfortunately I don't have the regex example available that caused this, but coming up hopefully you can take a guess what it looked like.

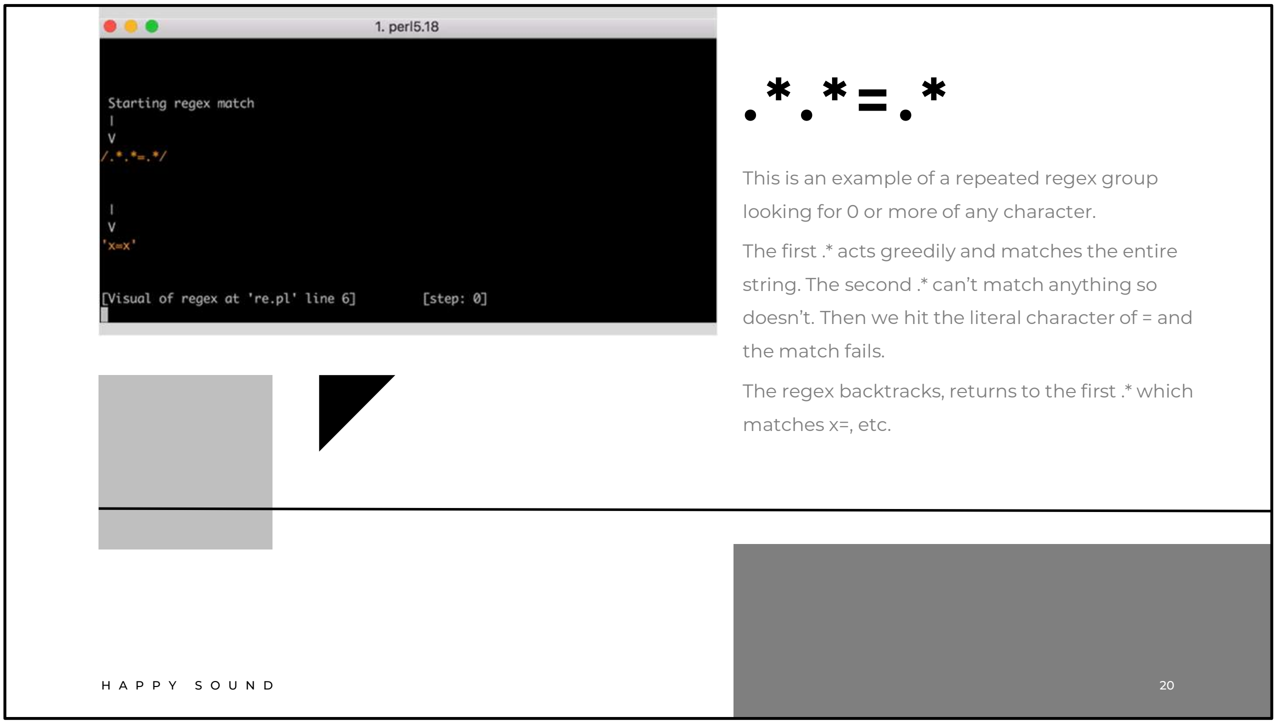
Catastrophic backtracking occurs when nested quantifiers are repeated, or groups themselves are repeated.

Catastrophic backtracking has been a problem since regex has been a thing. The solution to catastrophic backtracking has existed almost as long (first being identified and proposed in 1968). In more modern times, the .NET regex engine has been known to cause lots of problems (in the early 2000s, Hotmail devs facing 'catastrophic' issues… unfortunately, the blogs around that have been deleted).

As far as security is concerned, the primary problem that regex is going to cause you is denial of service – just like with the example that we saw with stack overflow.

At basically every stage of browsing the web you'll come up against a regex *somewhere* in the backend of your ops.

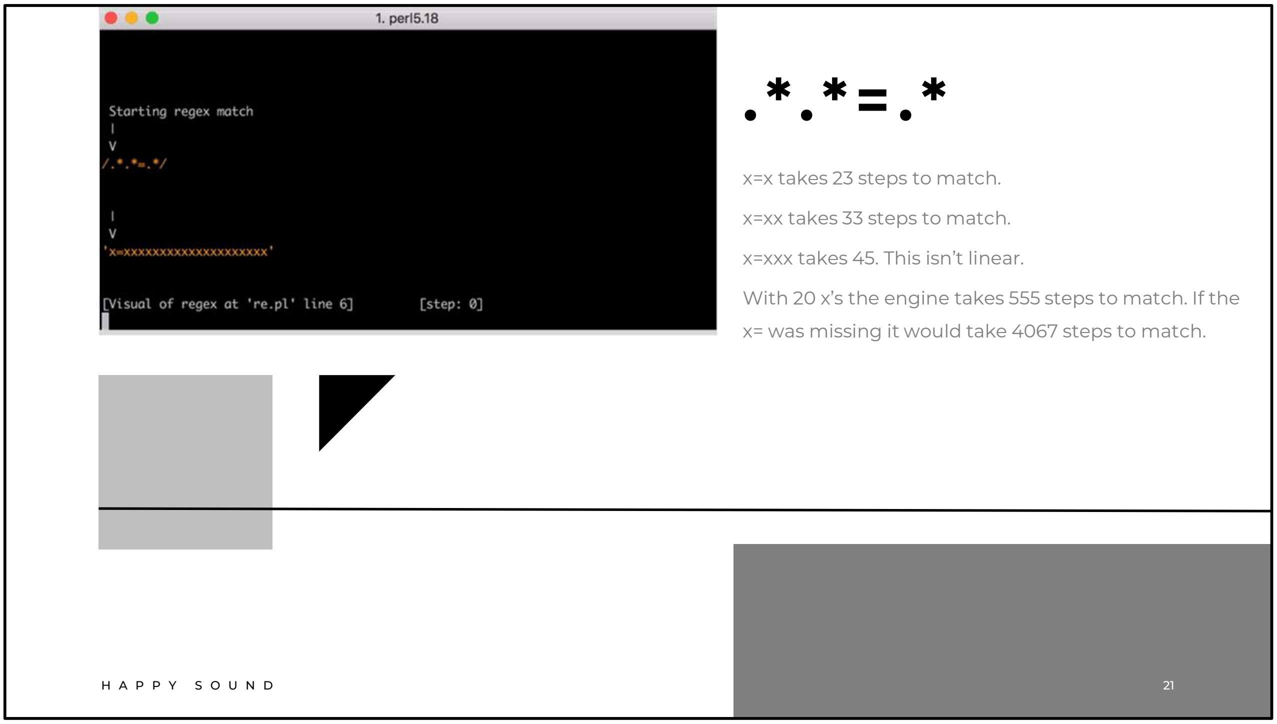
Evil regex – regex that can cause catastrophic backtracking – are usually relatively simple. They are almost always groups that are being repeated, or groups that are being repeated having further repetition or alternation inside.



.*.*=.* is essentially zero or more of any character, followed by zero or more of any character, followed by a literal =, followed by zero or more of any character.

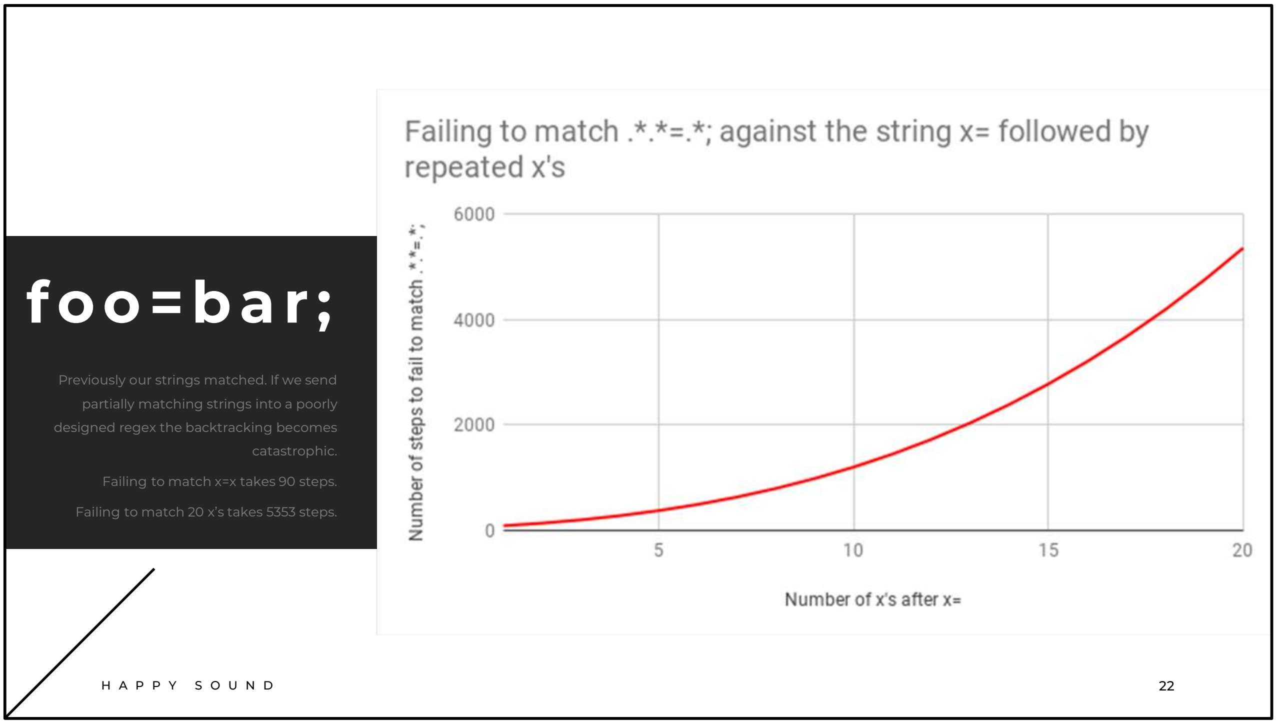
When looking at a simple match such as x=x it takes 23 steps.

The first step it takes is the .* matches the entire string. It can't match the second .*, it doesn't care, but when it gets to = it fails and backtracks.



If we extend the match to x=xx it takes 33 steps
When we get to 20x's it takes 555 steps.

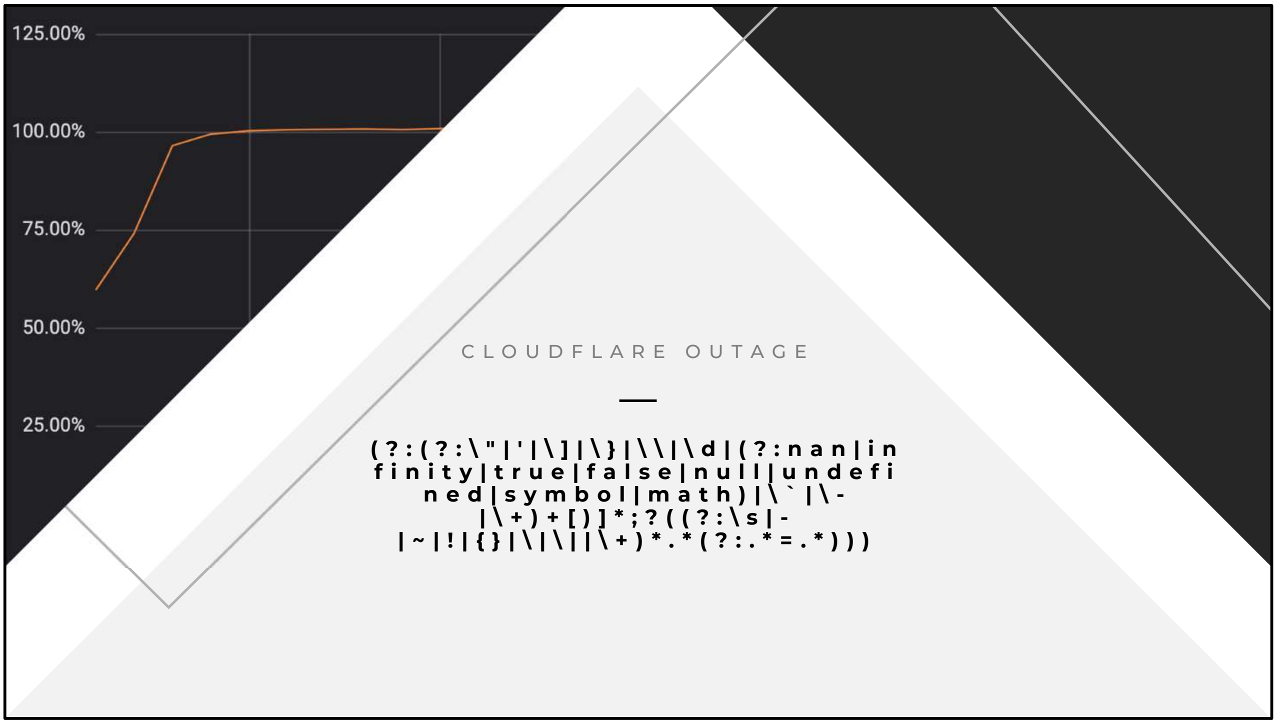
This is where we start getting to the realm of catastrophic backtracking.



If we add a semi-colon, such as trying to find the string foo=bar; our test string will always fail. But instead of taking 23 steps it takes 90 steps for x=x

For failing to match x= 20 x's it takes 5353 steps.


1. perl5.18
Starting regex match
|
V
/.*.*=.*;/
|
V
'x=xxxxxxxxxxxxxxxxxxxx'
[Visual of regex at 're.pl' line 6]
[step: 0]
HAPPY SOUND
23

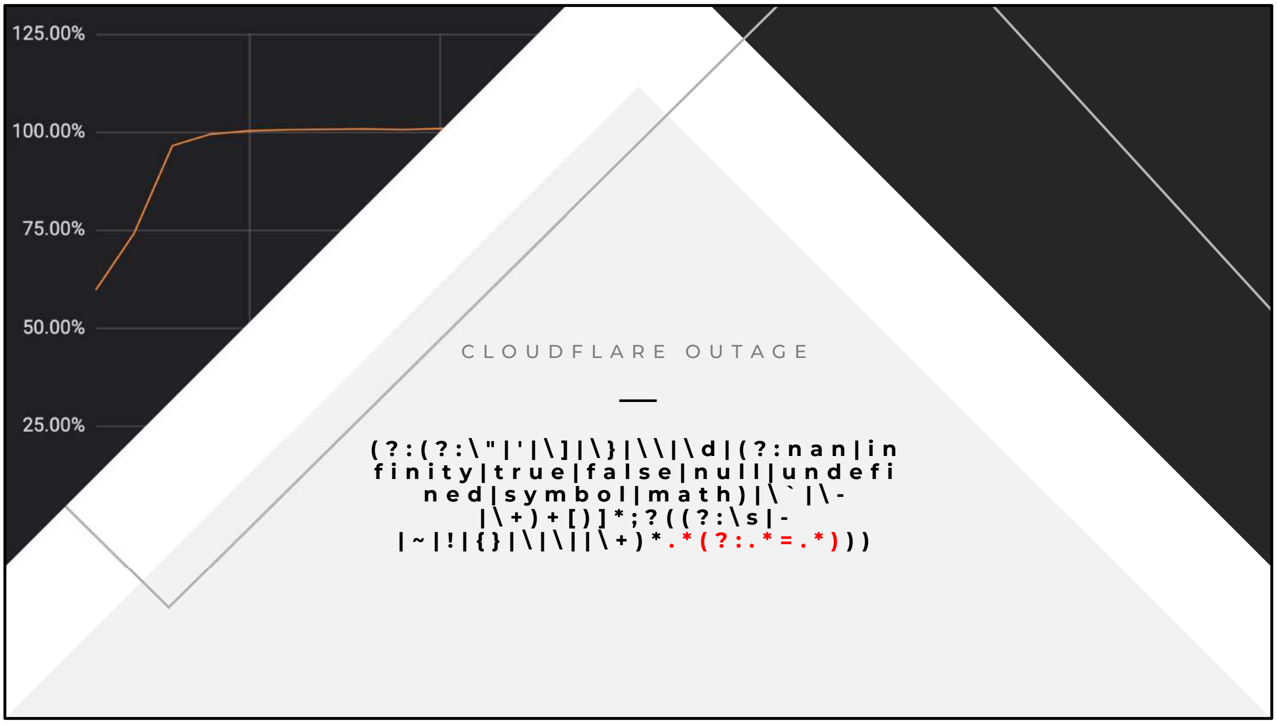


July 2, 2019 Cloudflare had a global outage which lasted an hour.

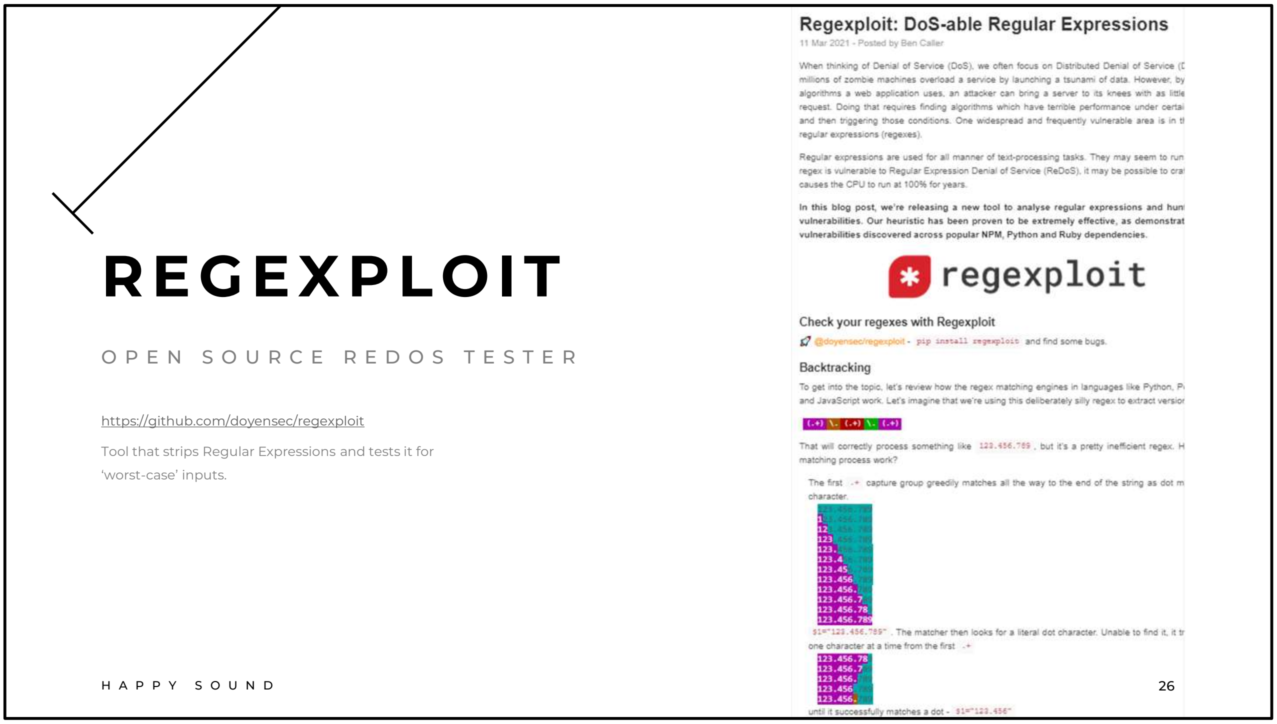
At 1342 an engineer working on the WAF teams deployed a minor change to look for XSS per a recent SharePoint vulnerability.

Global Services rapidly went down and after 20 minutes the Cloudflare team had identified that the WAF itself was the problem. It took another 10 minutes for them to terminate the WAF services globally.

Overall it took another 30 minutes to roll back changes to all devices globally and return WAF services globally.



We can see that the regex they deployed had the hallmarks of a regex which allows catastrophic backtracking.



ReDOS is simply, Denial of Service via sending malformed packets to a regex parser.

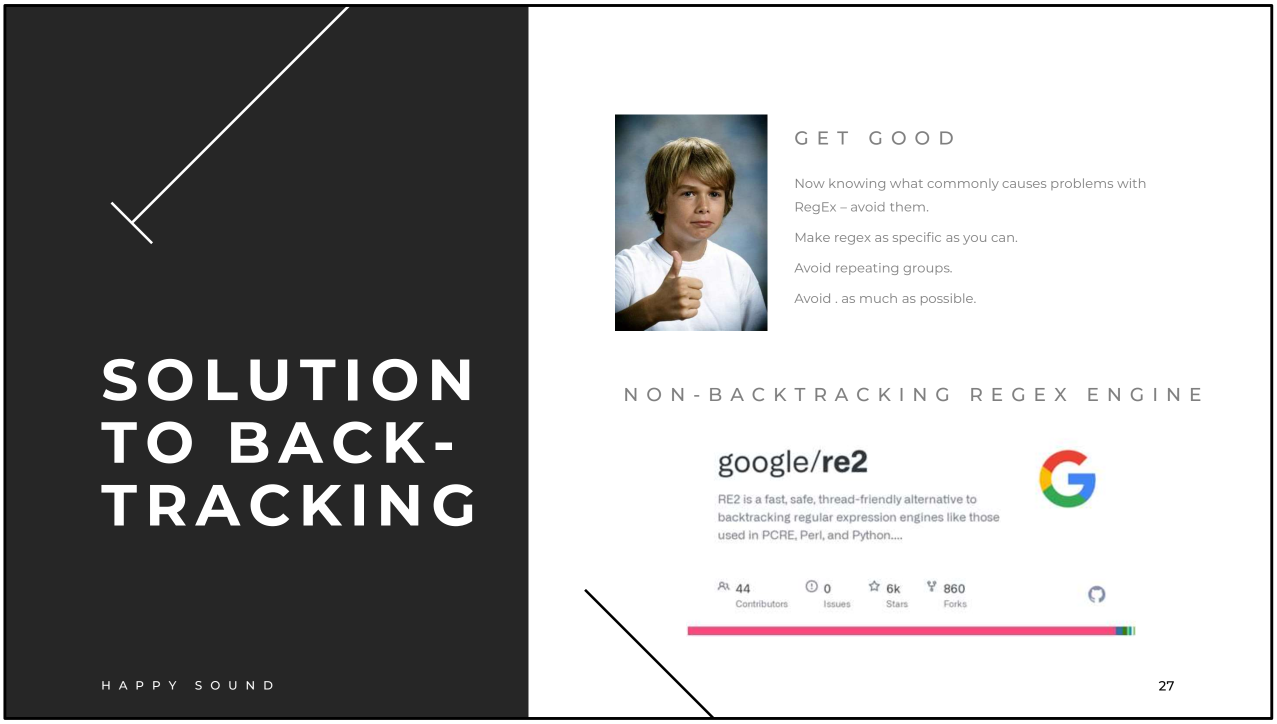
REDOS can be intentional by a malicious actor, or unintentional due to unexpected input (ie, 20000 whitespaces).

The DoyenSec crew created this tool to attempt to find Denial of Service attacks that can be exploited through bad regex.

One example was a bad date parser for cpython which had three consecutive \s* wrapped in optional groups. This meant that if you created a malicious cookie with 65506 spaces, it would take a week for the client to finish processing the header.

This effort gave them ten cves and many more unpublished bugs.

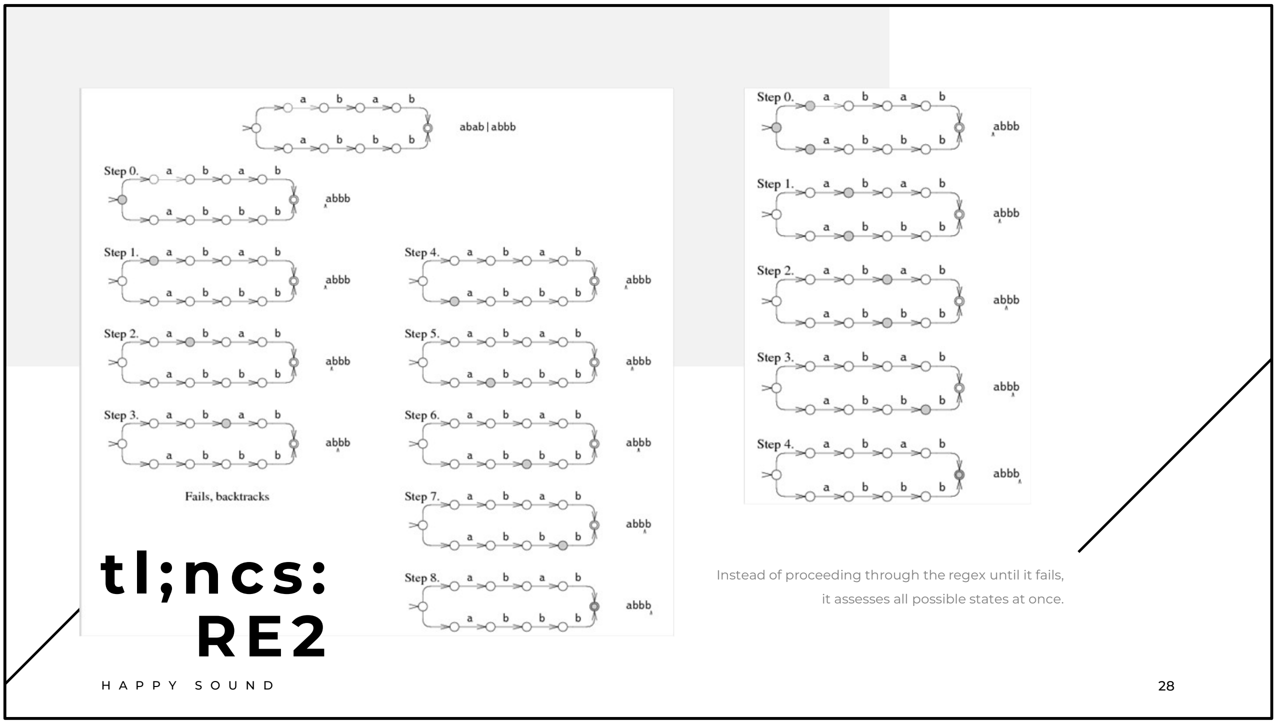
Read about it on their blog.



There are two simple solutions to avoiding catastrophic regex backtracking.

The first one is to simply git gud. I’ve explained the problem. Avoid repetitive matches. Try to avoid . and especially .* as much as possible.

Second: check out the google re2 engine. This engine doesn’t backtrack. What this means, however, is you cannot do back-references and can be slower for capturing operations. However, because RE2 does not backtrack it guarantees linear run-time, not exponential as we’ve seen.



This is where we start talking about NFA and DFA – Non-Deterministic, and Deterministic Finite Automaton.

The “Too Long; Not a Computer Scientist” of RE2 is that instead of proceeding through regex until it fails, then backtracking, is that it will assess all possible states at once.

Why isn’t this used more often? Well – most of the time, matching in a traditional NFA engine (such as PCRE) is simply ‘fast-enough’, and in those cases it’s usually faster than DFA.

HAPPY SOUND

FURTHER READING

Learn Regex: https://regexone.com/

Russ Cox – Regular Expression Matching can be Simple and Fast https://swtch.com/~rsc/regexp/regexp1.html

REDoS: https://blog.caller.xyz/user-agent-parsing-redos-cve-2020-5243/

DoyenSec Regexploit: https://blog.doyensec.com/2021/03/11/regexploit.html

Stack Overflow outage: https://stackstatus.net/post/147710624694/outage-postmortem-july-20-2016

Stack Overflow outage causing post: https://stackoverflow.com/questions/38484433/join-tiles-in-corona-sdk-into-one-word-for-a-breakout-game-grid


HAPPY SOUND
30