Regular Expression Denial of Service

(ReDoS Revisited)

Alex Roichman

Adar Weidman

CHECKMARX
SOURCE CODE ANALYSIS TECHNOLOGIES

OWASP
Israel 2009

The OWASP Foundation
http://www.owasp.org

Agenda

- DoS attack
- Regex and DoS - ReDoS
- Exploiting ReDoS: Why, What & How
- Leveraging ReDoS to Web attacks
  - Web application ReDoS
  - Client-side ReDoS
- Preventing ReDoS
- Conclusions and what next

DoS Attack

- The goal of Information Security is to preserve
  - Confidentiality
  - Integrity
  - Availability
- The final element in the CIA model, Availability, is often overlooked
- Attack on Availability - DoS
- DoS attack attempts to make a computer resource unavailable to its intended users

DoS Implication

- Whether DoS is dangerous or how to buy 100” TV for 1$
- DoS Attack vector:
  - Choose a public auction with a low start price
  - Submit your proposal
  - Prevent other users from submitting their proposals
  - Wait until the auction will be closed
  - Enjoy your new TV!

Brute-Force DoS

- Sending many requests such that the victim cannot respond to legitimate traffic, or responds so slowly as to be rendered effectively unavailable
- Flooding
- DDoS
- Amount of traffic is required to overload the server is big

Sophisticated DoS

- Hurting the weakest link of the system
- Application bugs
  - Buffer overflow
- Fragmentation of Data Structures
  - Hash Table
- Algorithm worst case
- Amount of traffic that is required to overload the server - little

From brute-force to Regex DoS

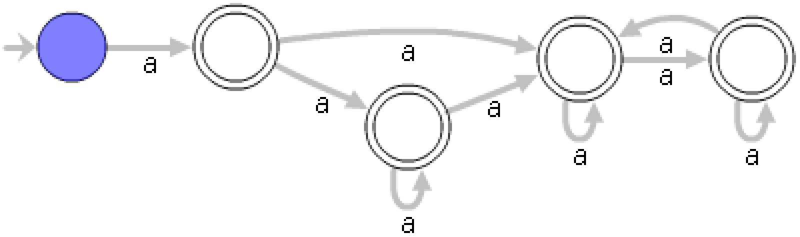
- Brute-force DoS is an old-fashion attack
  - It is network oriented
  - It can be easily detected/prevented by existing tools
  - It is hard to execute (great number of requests, zombies…)
- Sophisticated DoS by algorithm worst case is a new approach
  - It is application oriented
  - Hard to prevent/detect
  - Easy to execute (few request, no botnets)
- One kind of DoS is DoS by Regex or **ReDoS**

Regular Expressions

- Regular Expressions (Regexes) provide a concise and flexible means for identifying strings
- Regexes are written in a formal language that can be interpreted by a Regex engine
- Regexes are widely used
  - Text editors
  - Parsers/Interpreters/Compilers
  - Search engines
  - Text validations
  - Pattern matchers…

Regex engine algorithm

- The Regex engine builds Nondeterministic Finite Automata (NFA) for a given Regex
- For each input symbol NFA transitions to a new state until all input symbols have been consumed
- On an input symbol NFA may have several possible next states
- Example: (a+)+



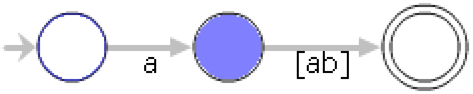
Regex Complexity

- In general case the number of different paths is exponential on the number of states
- Regex with backreferences
  - The problem is NP-complete, which was proven by Aho [1] – the best known algorithm is exponential
- There are better and worse Regex implementations, but even the best are exponential!

[1] A. V. Aho: Algorithms for finding patterns in strings

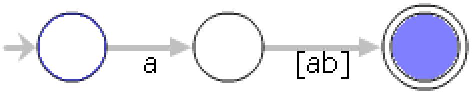
Regex Complexity Example - Linear

- Regex: a[ab]
- Payload: aaX
- First path



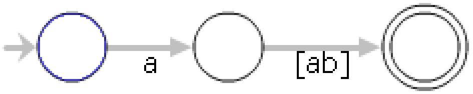
Regex Complexity Example - Linear

- Regex: a[ab]
- Payload: aaX
- First path



Regex Complexity Example - Linear

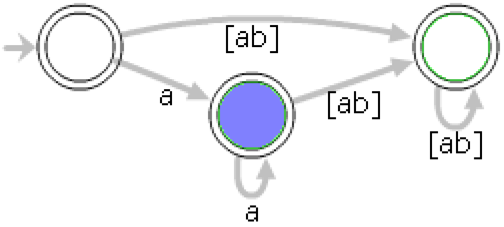
- Regex: a[ab]
- Payload: aaX
- First path



- Linear time

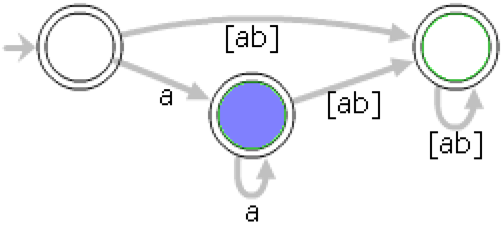
Regex Complexity Example - Quadratic

- Regex: a*[ab]*
- Payload: aaX
- First path



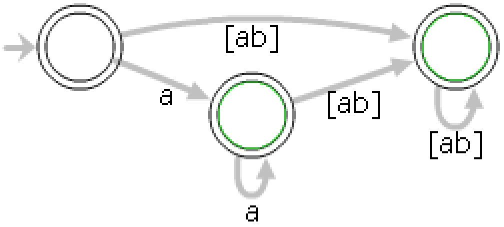
Regex Complexity Example - Quadratic

- Regex: a*[ab]*
- Payload: aaX
- First path



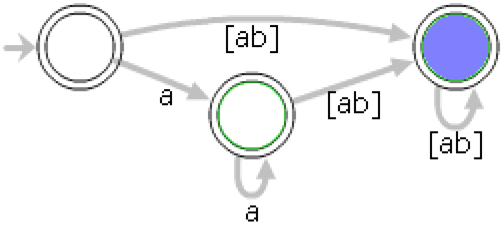
Regex Complexity Example - Quadratic

- Regex: a*[ab]*
- Payload: aaX
- First path



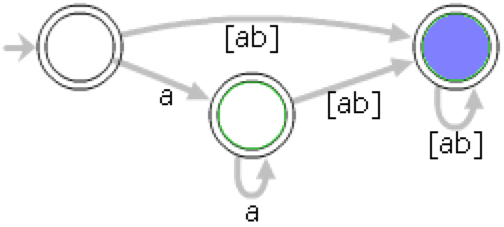
Regex Complexity Example - Quadratic

- Regex: a*[ab]*
- Payload: aaX
- Second path



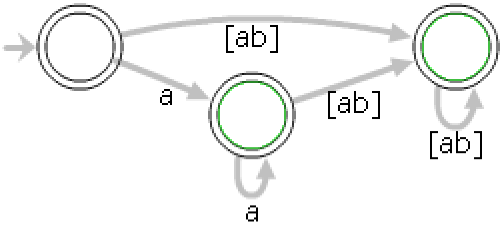
Regex Complexity Example - Quadratic

- Regex: a*[ab]*
- Payload: aaX
- Second path



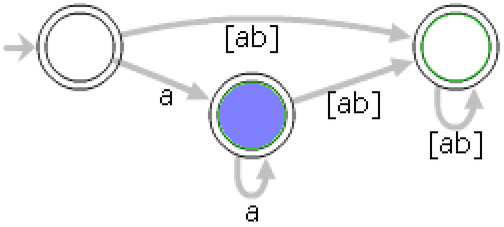
Regex Complexity Example - Quadratic

- Regex: a*[ab]*
- Payload: aaX
- Second path



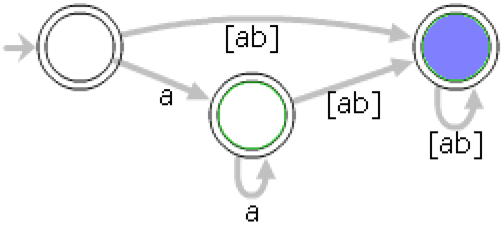
Regex Complexity Example - Quadratic

- Regex: a*[ab]*
- Payload: aaX
- Third path



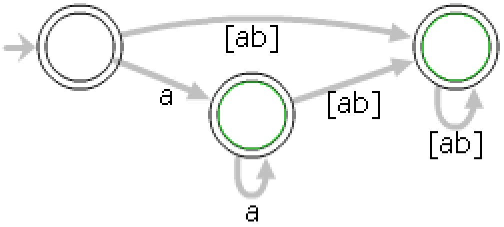
Regex Complexity Example - Quadratic

- Regex: a*[ab]*
- Payload: aaX
- Third path



Regex Complexity Example - Quadratic

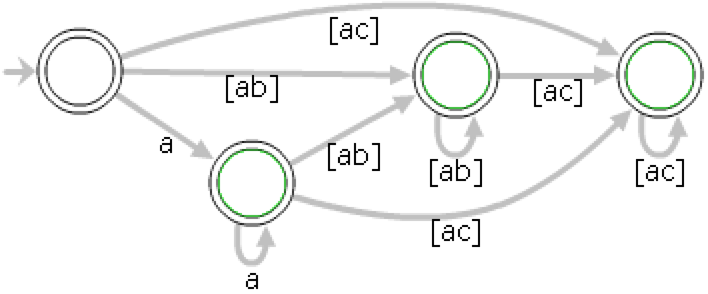
- Regex: a*[ab]*
- Payload: aaX
- Third path



- Quadratic time

Regex Complexity Example - Cubic

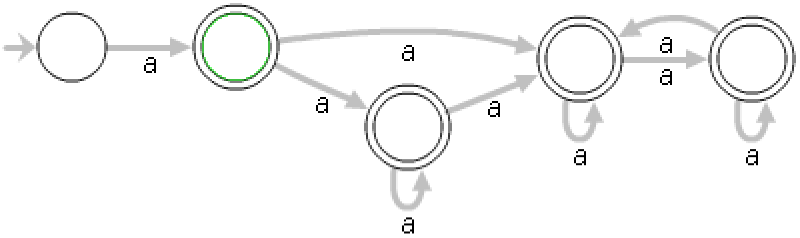
- Regex: a*[ab]*[ac]*
- Payload: aaX
- Seven paths



- Cubic time

Regex Complexity Example - Exponential

- Regex: (a*)*
- Payload: aaX



- Exponential time

ReDoS on the Web

- If unsafe Regexes run on inputs which cannot be matched, then the Regex engine is stuck
- The fact that some evil Regexes may result on DoS was mentioned in 2003 by [2]
- In our research we want to revisit an old attack and show how we can leverage it on the Web
- The art of attacking the Web by ReDoS is by finding inputs which cannot be matched by the above Regexes and on these Regexes a Regex-based Web systems will stuck

[2] http://www.cs.rice.edu/~scrosby/hash/slides/USENIX-RegexpWIP.2.ppt

Evil Regex Patterns

- (a+)+
- (a*)*
- (a|aa)+
- (a|a?)+
- (.*a){x} | for x > 10

Payload: aaaaaaaaaaaX

Real examples of ReDoS

- OWASP Validation Regex Repository
  - Person Name
    - Regex: ^[a-zA-Z]+(([\'\,\.\-][a-zA-Z])?[a-zA-Z]*)*$
    - Payload: aaaaaaaaaaaaaaaaaaaaaaaaaaaa!
  - Java Classname
    - Regex: ^(([a-z])+.)+[A-Z]([a-z])+$
    - Payload: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa!

Real examples of ReDoS

- Regex Library
  - Email Validation
    - Regex: ^([0-9a-zA-Z]([-.\w]*[0-9a-zA-Z])*@(([0-9a-zA-Z])+([-\w]*[0-9a-zA-Z])*\.)+[a-zA-Z]{2,9})$
    - Payload: a@aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa!
  - Multiple Email address validation
    - Regex: ^[a-zA-Z]+(([\'\,\.\-][a-zA-Z])?[a-zA-Z]*)*\s+&lt;(\w[-._\w]*\w@\w[-._\w]*\w\.\w{2,3})&gt;$|^(\w[-._\w]*\w@\w[-._\w]*\w\.\w{2,3})$
    - Payload: aaaaaaaaaaaaaaaaaaaaaaaa!
  - Decimal validator
    - Regex: ^\d*[0-9](|.\d*[0-9]|)*$
    - Payload: 1111111111111111111111111!
  - Pattern Matcher
    - Regex: ^([a-z0-9]+([\-a-z0-9]*[a-z0-9]+)?\.){0,}([a-z0-9]+([\-a-z0-9]*[a-z0-9]+)?){1,63}(\.[a-z0-9]{2,7})+$
    - Payload: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa!

Exploiting ReDoS: Why

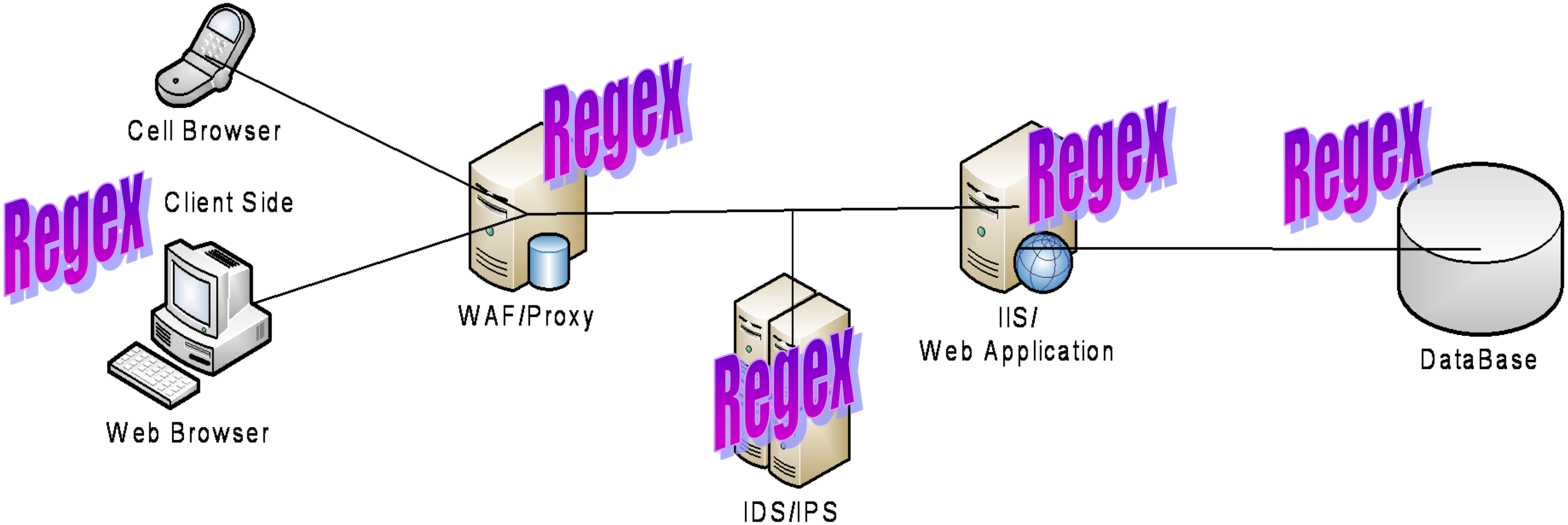
- The art of writing robust Regexes is obscure and difficult
- Programmers are not aware of Regex threats
- Security experts are not aware of DoS on regexes
- There are no tools for ReDoS-safety validating
- By bringing a Regex engine to its worst exponential case, an attacker can easily exploit DoS.

Exploiting ReDoS: How

- There are two ways to ReDoS a system:
  - Crafting a special input for an existing system Regex
    - Build a string for which a system Regex has no match and on this string a Regex machine will try all available paths until it rejects the string
      - Regex: (a+)+
      - Payload: aaaaaaaaX
  - Injecting a Regex in case a system builds it dynamically
    - Build Regex with many paths which will “stack-in” on a system string by using all these paths until it rejects the string
      - Regex: (a+)+X
      - Payload: aaaaaaa

Exploiting ReDoS: What

- Regexes are ubiquitous now – web is Regex-based



- In this presentation we will discuss ReDoS attacks on:
  - Web application
  - Client-side

Web application ReDoS

- Regular expressions are widely used for implementing application validation rules.
- There are two main strategies for validating inputs by Regexes:
  - Accept known good. In such a case Regex should begin with “^” and end with “$” character to validate an entire input and not only part of it.
  - Reject known bad. In such a case Regex can be used to identify an attack fingerprints.

Web application ReDoS

- Crafting malicious input for a given Regex
  - Programmers are not aware of evil Regexes
  - QA generally check for valid inputs, attackers exploit invalid inputs on which Regex engine will try all existing paths until it reject the input
  - There are no dynamic tools for Regex evaluation
  - In many cases the attack is simple and not blind:
    - Many applications are open source
    - The same Regex appears both in client-side and in server-side

Web application ReDoS

- Application ReDoS attack vector 1:
  - Open a JavaScript
  - Find evil Regex
  - Craft a malicious input for a found Regex
  - Submit a valid value via intercepting proxy and change the request to contain a malicious input
  - You are done!

Web application ReDoS

- Crafting malicious Regex for a given string.
  - Many applications receive a search key in format of Regex
  - Many applications build Regex by concatenating user inputs
  - Regex Injection [3] like other injections is a common application vulnerability

[3] C. Wenz: Regular Expression Injection

Web application ReDoS

- Application ReDoS attack vector 2:
  - Find a Regex injection vulnerable input by submitting an invalid escape sequence like “\m”
  - If the following message is received: “invalid escape sequence”, then there is Regex injection
  - Submit “(a+)+\u0001”
  - You are done!

Web application ReDoS Example

- DataVault:
  - Regex: ^\[(,.*)*\]$
  - Payload: [,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
- WinFormsAdvansed:
  - Regex: \A([A-Z,a-z]*\s?[0-9]*[A-Z,a-z]*)*\Z
  - Payload: aaaaaaaaaaaaaaaaaa!
- EntLib
  - Regex: ^([^\"]+)(?:\\([^\"]+))*$
  - Payload: \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\"

Client-side ReDoS

- Internet browsers spend many efforts to prevent DoS on them.
- Between issues that browsers prevent:
  - Infinite loops
  - Long iterative statements
  - Endless recursions
- But what about Regex?

Client-side ReDoS

- New multiple vendor Web Browser JavaScript Denial Of Service
- Relevant for all Java/JavaScript based browsers
- Relevant also for all cellular devices with a browsing ability
- DoS on a cellular device is a serious attack

Client-side ReDoS

- Browsers ReDoS attack vector:
  - Deploy a page containing the following JavaScript code:

```
<html>
  <script language='jscript'>
    myregexp = new RegExp(/^(a+)+$/);
    mymatch = myregexp.exec("aaaaaaaaaaaaaaaaaaaaaaaab");
  </script>
</html>
```

  - Trick a victim to browse this page
  - You are done!

Preventing ReDoS

- ReDoS vulnerability is serious so we should be able to prevent/detect it
- Any Regex should be checked for ReDoS safety prior to using it
- Dynamically built user input-based Regex should not be used
- The following tools can be used for Regex safety testing:
  - Dynamic Regex testing, pen testing/fuzzing
  - Static Regex code analyzer

ReDoS and dynamic tools

- Prevention vector 1:
  - Try to penetrate the system with different inputs
  - Check a response time of the system, if it increases- try to repeat characters of a given input
  - If a response time get slow – you are ReDoSed!
- Prevention vector 2:
  - Try to inject an invalid escape sequence like “\m”
  - If a response is different from a response on a valid input – you are probably ReDoSed

ReDoS and static code analysis

- Prevention vector 3:
  - Analyze the source code and look for Regex
  - Check each found Regex whether it contains an evil patterns or can be data-influenced by a user input
  - If it does – you are ReDoSed!

Conclusions

- The web is Regex-based
- The border between safe and unsafe Regex is very ambiguous
- In our research we wanted to revisit ReDos and to expose the problem to the application security community
- In our research we show that the Regex worst (exponential) case may be easily leveraged to DoS attacks on the web

What next?

- Extra research is required in the following fields:
  - Current state assessment – to what extent we are vulnerable to ReDoS
  - Finding additional evil Regex patterns
  - Finding additional attack vectors on evil Regex
  - Developing tools for dynamic Regex evaluation
  - Developing tools for static Regex evaluation