Regular Expression Denial of Service

Alex Roichman

Chief Architect, Checkmarx

Adar Weidman

Senior Programmer, Checkmarx

Agenda

- DoS attack
- Regex and DoS - ReDoS
- Exploiting ReDoS: Why, Where & How
- Leveraging ReDoS to Web attacks
  - Server-side ReDoS
  - Client-side ReDoS
- Preventing ReDoS
- Conclusions

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

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
- 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…)
  - Large amount of traffic is required to overload the server

Sophisticated DoS

- Hurting the weakest link of the system
- Application bugs
  - Buffer overflow
- Fragmentation of Data Structures
  - Hash Table
- Algorithm worst case
- Sophisticated DoS is a new approach
  - It is application oriented
  - Hard to prevent/detect
  - Easy to execute (few requests, no botnets)
  - Amount of traffic that is required to overload the server - little

From Sophisticated DoS to Regex DoS

- One kind of sophisticated DoS is DoS by Regex or **ReDoS**
- It is believed that Regex performance is fast, but the truth is that the Regex worst case is exponential
- In this presentation we will show how an attacker can easily exploit the Regex worst case and cause an application DoS
- We will show how an application can be ReDoSed by sending only one small message

ReDoS on the Web

- The fact that some evil Regexes may result on DoS was mentioned in 2003 by [1]
- In our research we want to revisit an old attack and show how we can leverage it on the Web
- If unsafe Regexes run on inputs which cannot be matched, then the Regex engine is stuck
- 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 get stuck

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

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 by
  - Text editors
  - Parsers/Interpreters/Compilers
  - Search engines
  - Text validations
  - Pattern matchers…

Regex Implementations

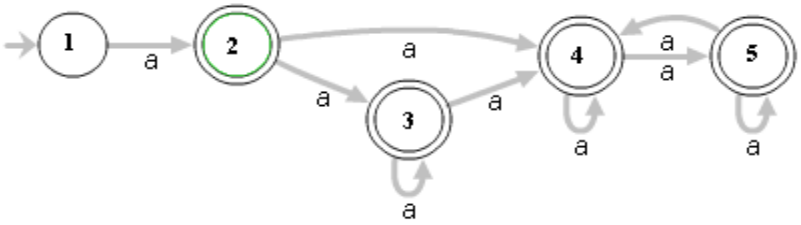
- There are at least two different algorithms that decide if a given string matches a regular expression:
  - Perl-based
  - NFA-based
- NFA-based implementation is very simple and allows the use of backreferences
- Many environments like .NET, Java, JavaScript etc. use NFA-based implementation

Regex NFA-based 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
- Regex deterministic algorithm tries all paths of NFA one after the other until it reaches an accepting state (match) or all paths are tried (no match)

Regex Complexity

- In a general case the number of different paths is exponential for input length
- Regex: ^(a+)+$
- Payload: “aaaaX”



- # of different paths: 16
- Regex worst case is **exponential**
- How many paths do we have for “aaaaaaaaaaX”? 1024
- And for “aaaaaaaaaaaaaaaaaaaaX”?...

Evil Regex Patterns

- Regex is called evil if it can be stuck on specially crafted input
- Each evil Regex pattern should contain:
  - Grouping construct with repetition
  - Inside the repeated group there should appear
    - Repetition
    - Alternation with overlapping
- Evil Regex pattern examples
  1. (a+)+
  2. ([a-zA-Z]+)*
  3. (a|aa)+
  4. (a|a?)+
  5. (.*a){x} | for x > 10

Payload: “aaaaaaaaaaaaaaaaaaX”

Real examples of ReDoS

- OWASP Validation Regex Repository [2]
  - Person Name
    - Regex: ^[a-zA-Z]+(([\'\,\.\-][a-zA-Z])?[a-zA-Z]*)*$
    - Payload: “aaaaaaaaaaaaaaaaaaaaaaaaaaa!”
  - Java Classname
    - Regex: ^(([a-z])+.)+[A-Z]([a-z])+$
    - Payload: “aaaaaaaaaaaaaaaaaaaaaaaaaaaaa!”

[2] http://www.owasp.org/index.php/OWASP_Validation_Regex_Repository

Real examples of ReDoS

- Regex Library [3]
  - 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@aaaaaaaaaaaaaaaaaaaaaaaaaaaaa!
  - Multiple Email address validation
    - Regex: ^[a-zA-Z]+((['\,\.\-][a-zA-Z])?[a-zA-Z]*)*\s+<(\w[-._\w]*\w@\w[-._\w]*\w\.\w{2,3})>$|^(\w[-._\w]*\w@\w[-._\w]*\w\.\w{2,3})$
    - Payload: aaaaaaaaaaaaaaaaaaaaaa!
  - Decimal validator
    - Regex: ^\d*[0-9](|.\d*[0-9]|)*$
    - Payload: 11111111111111111111111!
  - 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: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa!

[3] http://regexlib.com/

Exploiting ReDoS: Why

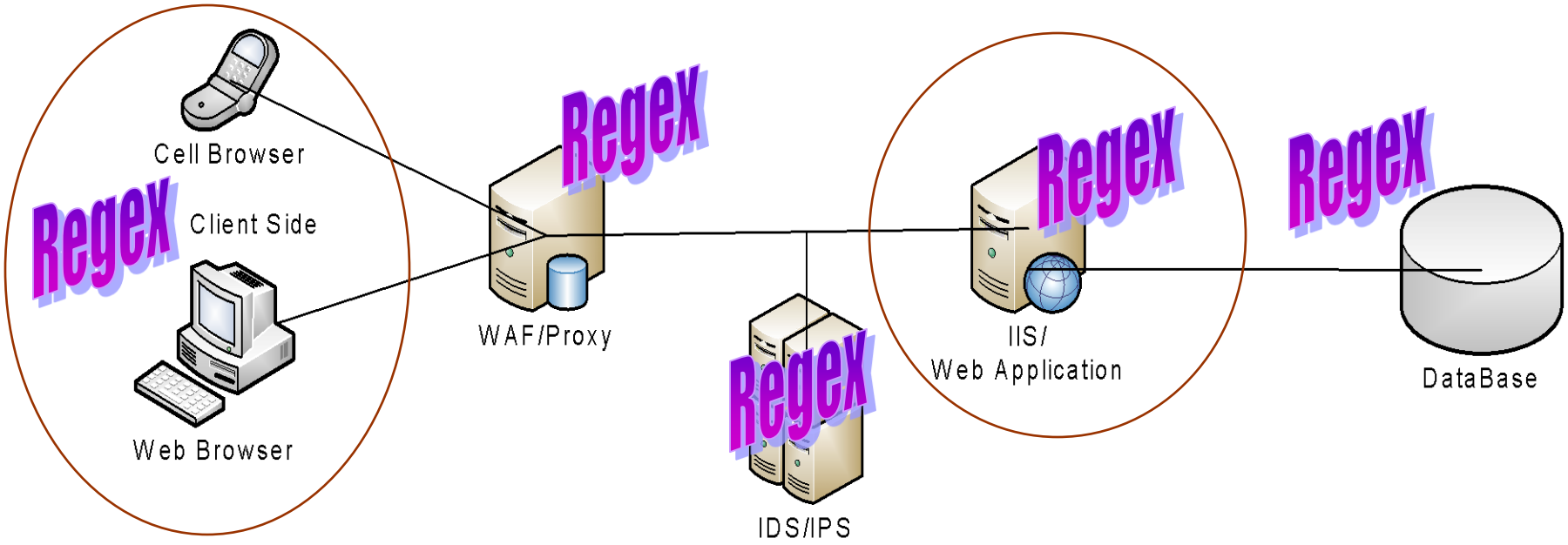
- The art of writing robust Regexes is obscure and difficult
- Programmers are not aware of Regex threats
- QA generally checks for valid inputs, attackers exploit invalid inputs on which a Regex engine will try all existing paths until it rejects the input
- Security experts are not aware of DoS on Regexes
- There are no tools for ReDoS-safety validation
- 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 if a system builds it dynamically
    - Build a Regex with many paths which will be “stuck-in” on a system string by using all these paths until it rejects the string
      - Payload : (.+)+\u001

Exploiting ReDoS: Where

- Regexes are ubiquitous now – the web is Regex-based



Web application ReDoS – Regex Validations

- Regular expressions are widely used for implementing application validation rules.
- There are two main strategies for validating inputs by Regexes:
  - Accept known good-
    - Regex should begin with “^” and end with “$” character to validate an entire input and not only part of it
    - Very tight Regex will cause False Positives (DoS for a legal user!)
  - Reject known bad-
    - Regex can be used to identify an attack fingerprint
    - Relaxed Regex can cause False Negatives

Web application ReDoS – Malicious Inputs

- Crafting malicious input for a given Regex
- Blind attack – Regex is unknown to an attacker:
  - Try to understand which Regex can be used for a selected input
  - Try to divide Regex into groups
  - For each group try to find a string which cannot be matched
- Non-blind attack - Many applications are open source; many times the same Regex appears both in client-side and in server-side:
  - Understand a given Regex and build a malicious input

Web application ReDoS – Attack 1

- A server side application can be stuck in when client-side validations are backed-up on a server side
- Application ReDoS attack vector 1:
  - Open a source of Html
  - Find evil Regex in JavaScript
  - 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 – Malicious Regexs

- 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** [4] like other injections is a common application vulnerability
  - Regex Injection can be used to stuck an application

[4] C. Wenz: Regular Expression Injection

Web application ReDoS – Attack 2

- 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 a Regex injection
  - Submit “(.+)+\u0001”
  - You are done!

Google CodeSearch Hacking

- Google CodeSearch involves using advanced operators and Regexes in the Google search engine to locate specific strings of text within open sources [5]:
- Meta-Regex is a Regex which can be used to find evil Regexes in a source code:
  - Regex.+\(\.*\)\+
  - Regex.+\(.\.*\)*
- **Google CodeSearch Hacking –** involves using meta-Regexes to find evil Regexes in open sources

[5] http://www.google.com/codesearch

Web application ReDoS Example

- DataVault [6]:
  - Regex: ^\[(,.*)*\]$
  - Payload: [,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
- WinFormsAdvanced [7]:
  - Regex: \A([A-Z,a-z]*\s?[0-9]*[A-Z,a-z]*)*\Z
  - Payload: aaaaaaaaaaaaaaaaa!
- EntLib [8]:
  - Regex: ^([^\"]+)(?:\\([^\"]+))*$
  - Payload: \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\"

[6] http://www.google.com/codesearch/p?hl=en&sa=N&cd=3&ct=rc#4QmZNJ8GGhI/trunk/DataVault.Tesla/Impl/TypeSystem/AssociationHelper.cs
[7] http://www.google.com/codesearch/p?hl=en&sa=N&cd=1&ct=rc#nVoRdQ_MJpE/Zoran/WinFormsAdvansed/RegeularDataToXML/Form1.cs
[8] http://www.google.com/codesearch/p?hl=en&sa=N&cd=4&ct=rc#Y_Z6zi1FBas/Blocks/Common/Src/Configuration/Manageability/Adm/AdmContentBuilder.cs

Client-side ReDoS

- Internet browsers spend tremendous efforts to prevent DoS.
- Among the issues that browsers prevent:
  - Infinite loops
  - Long iterative statements
  - Endless recursions
- But what about Regex?
- Relevant for Java/JavaScript based browsers

Client-side ReDoS – Browser ReDoS

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

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

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

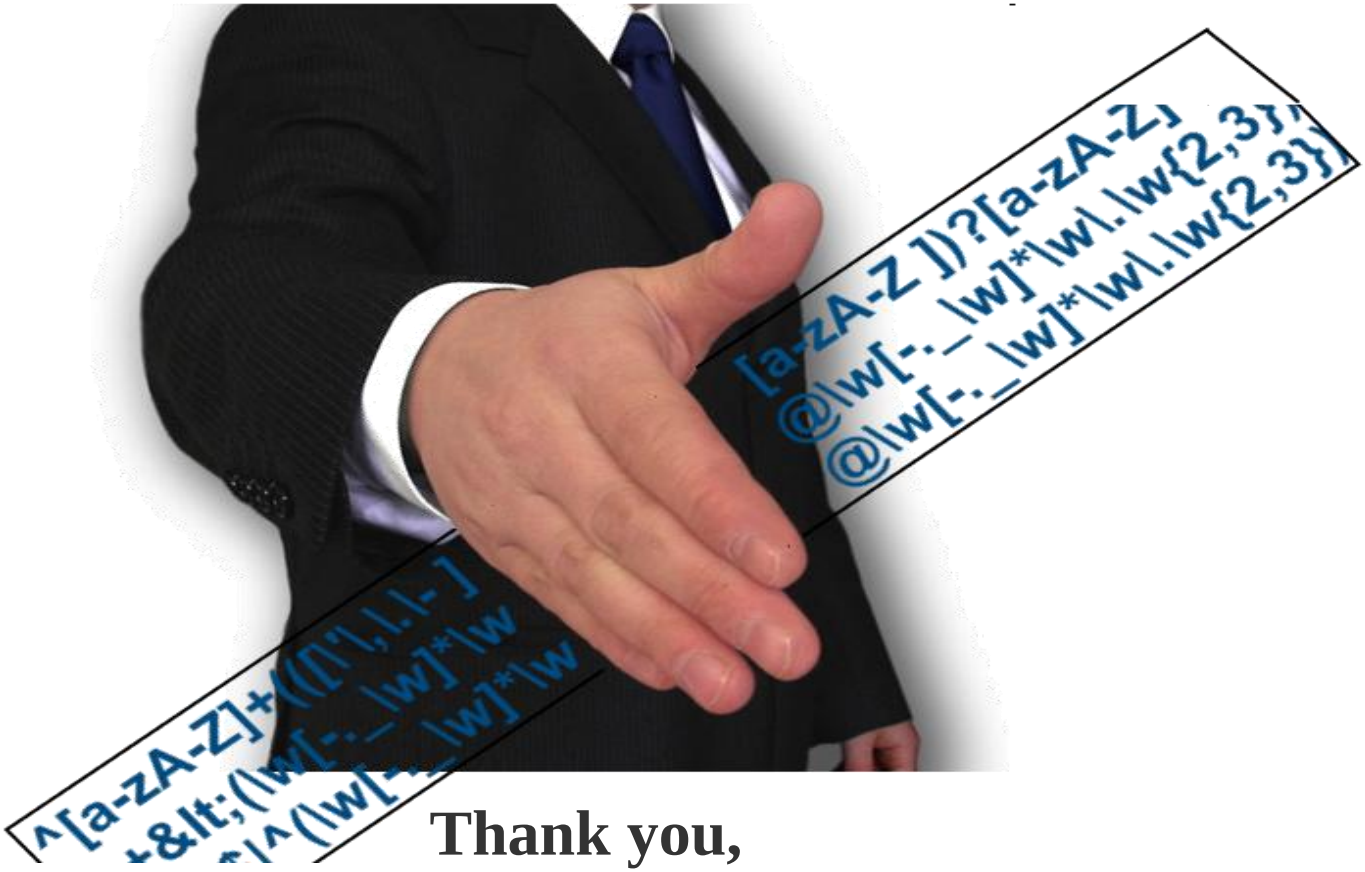
Preventing ReDoS

- ReDoS vulnerability is serious so we should be able to prevent/detect it
- Dynamically built input-based Regex should not be used or should use appropriate sanitizations
- Any Regex should be checked for ReDoS safety prior to using it
- The following tools should be developed for Regex safety testing:
  - Dynamic Regex testing, pen testing/fuzzing
  - Static code analysis tools for unsafe-Regex detection

Conclusions

- The web is Regex-based
- The border between safe and unsafe Regex is very ambiguous
- In our research we show that the Regex's worst exponential case may be easily leveraged to DoS attacks on the web
- In our research we revisited ReDoS and tried:
  - to expose the problem to the application security community
  - to encourage development of Regex-safe methodologies and tools

ReDoS – Q&A



Thank you,

Alex Roichman

Alexr@Checkmarx.com