WIKIPEDIA

ReDoS

The **regular expression denial of service** (**ReDoS**)[1] is an algorithmic complexity attack that produces a denial-of-service by providing a regular expression that takes a very long time to evaluate. The attack exploits the fact that most regular expression implementations have exponential time worst case complexity: the time taken can grow exponentially in relation to input size. An attacker can thus cause a program to spend an unbounded amount of time processing by providing such a regular expression, either slowing down or becoming unresponsive.[2][3]

Contents

Description

Regular expression matching can be done by building a finite-state automaton. Regular expressions can be easily converted to nondeterministic automata (NFAs), in which for each state and input symbol there may be several possible next states. After building the automaton, several possibilities exist:

- the engine may convert it to a deterministic finite-state automaton (DFA) and run the input through the result;
- the engine may try one by one all the possible paths until a match is found or all the paths are tried and fail ("backtracking").[4][5]
- the engine may consider all possible paths through the nondeterministic automaton in parallel;
- the engine may convert the nondeterministic automaton to a DFA lazily (*i.e.*, on the fly, during the match).

Of the above algorithms, the first two are problematic. The first is problematic because a deterministic automaton could have up to 2^m states where m is the number of states in the nondeterministic automaton; thus, the conversion from NFA to DFA may take exponential time. The second is problematic because a nondeterministic automaton could have an exponential number of paths of length n, so that walking through an input of length n will also take exponential time.[6] The last two algorithms, however, do not exhibit pathological behavior.

Note that for non-pathological regular expressions the problematic algorithms are usually fast, and in practice one can expect them to "compile" a regular expression in O(m) time and match it in O(n) time; instead, simulation of an NFA and lazy computation of the DFA have O(m^2n) worst-case complexity.[7] Regular expression denial of service occurs when these expectations are applied to regular expressions provided by the user, and malicious regular expressions provided by the user trigger the worst-case complexity of the regular expression matcher.

While regex algorithms can be written in an efficient way, most regular expression engines in existence extend the regular expression languages with additional constructs that cannot always be solved efficiently. Such extended patterns essentially force the implementation of regular expression in most programming languages to use backtracking.

Examples

Malicious regexes

Malicious regexes that get stuck on crafted input can be different depending on the regular expression matcher that is under attack. For backtracking matchers, they occur whenever these factors occur:[8]

- the regular expression applies repetition ("+", "*") to a complex subexpression;
- for the repeated subexpression, there exists a match which is also a suffix of another valid match.

The second condition is best explained with an example: in the regular expression `(a[ab]*)+`, both "a" and "aa" can match the repeated subexpression `a[ab]*`. Therefore, after matching "a", the nondeterministic automaton may try a new match of `a[ab]*` or a new match of `a`. If the input has many consecutive "a"s, each of them will double the number of possible paths through the automaton. Examples of malicious regexes include the following:

- `(a+)+`
- `([a-zA-Z]+)*`
- `(a|aa)+`
- `(a|a?)+`
- `(.*a){x}` for x > 10

All the above are susceptible to the input `aaaaaaaaaaaaaaaaaaaaaaaa!`. (The minimum input length might change slightly, when using faster or slower machines.)

Other patterns may not cause an exponential behavior, but for long enough inputs (a few hundreds of characters, usually) they could still cause long elaboration times. An example of such a pattern is "`a*b?a*x`", the input being an arbitrarily long sequence of "a"s. Such a pattern may also cause backtracking matchers to hang.

Vulnerable regexes in online repositories

So-called "evil" or malicious regexes have been found in online regular expression repositories. Note that it is enough to find an evil *sub*expression in order to attack the full regex:

1. RegExLib, id=1757 (email validation) (http://regexlib.com/REDetails.aspx?regexp_id=1757) - see red part, which is an Evil Regex
   `^([a-zA-Z0-9])(([\-.]|[_]+)?([a-zA-Z0-9]+))*(@){1}[a-z0-9]+[.]{1}(([a-z]{2,3})|([a-z]{2,3}[.]{1}[a-z]{2,3}))$`
2. OWASP Validation Regex Repository (http://www.owasp.org/index.php/OWASP_Validation_Regex_Repository), Java Classname - see red part, which is an Evil Regex
   `^(([a-z])+.)+[A-Z]([a-z])+$`

These two examples are also susceptible to the input `aaaaaaaaaaaaaaaaaaaaaaaa!`.

Attacks

If a regex itself is affected by a user input, the attacker can inject a malicious regex and make the system vulnerable. Therefore, in most cases, regular expression denial of service can be avoided by removing the possibility for the user to execute arbitrary patterns on the server. In this case, web applications and databases are the main vulnerable applications. Alternatively, a malicious page could hang the user's web browser or cause it to use arbitrary amounts of memory.

However, some of the examples in the above paragraphs are considerably less "artificial" than the others; thus, they demonstrate how a vulnerable regexes may be used as a result of programming mistakes. In this case e-mail scanners and intrusion detection systems could also be vulnerable. Fortunately, in most cases the problematic regular expressions can be rewritten as "non-evil" patterns. For example, `(.*a){x}` can be rewritten to `([^a]*a){x,}`.

In the case of a web application, the programmer may use the same regular expression to validate input on both the client and the server side of the system. An attacker could inspect the client code, looking for evil regular expressions, and send crafted input directly to the web server in order to hang it.

See also

- Denial-of-service attack
- Pentium F00F bug
- Cyberwarfare
- Low Orbit Ion Cannon
- High Orbit Ion Cannon

References

1. OWASP (2010-02-10). "Regex Denial of Service" (http://www.owasp.org/index.php/Regular_expression_Denial_of_Service_-_ReDoS). Retrieved 2010-04-16.
2. RiverStar Software (2010-01-18). "Security Bulletin: Caution Using Regular Expressions" (https://www.riverstarsoftware.com/kb/article/security-bulletin-caution-using-regular-expressions-17.html). Retrieved 2010-04-16.
3. Ristic, Ivan (2010-03-15). *ModSecurity Handbook* (https://www.feistyduck.com/books/modsecurity-handbook/index.html). London, UK: Feisty Duck Ltd. p. 173. ISBN 978-1-907117-02-2.
4. Crosby and Wallach, Usenix Security (2003). "Regular Expression Denial Of Service" (http://www.cs.rice.edu/~scrosby/hash/slides/USENIX-RegexpWIP.2.ppt). Retrieved 2010-01-13.
5. Bryan Sullivan (http://msdn.microsoft.com/en-us/magazine/ee532098.aspx?sdmr=BryanSullivan&sdmi=authors), MSDN Magazine (2010-05-03). "Regular Expression Denial of Service Attacks and Defenses" (http://msdn.microsoft.com/en-au/magazine/ff646973.aspx). Retrieved 2010-05-06.
6. Kirrage, J.; Rathnayake, A.; Thielecke, H. (2013). "Static Analysis for Regular Expression Denial-of-Service Attacks". *Network and System Security*. Madrid, Spain: Springer. pp. 135–148. arXiv:1301.0849 (https://arxiv.org/abs/1301.0849). doi:10.1007/978-3-642-38631-2_11 (https://doi.org/10.1007%2F978-3-642-38631-2_11).
7. Lazy computation of the DFA can usually reach the speed of deterministic automatons while keeping worst case behavior similar to simulation of an NFA. However, it is considerably more complex to implement and can use more memory.
8. Jim Manico and Adar Weidman (2009-12-07). "OWASP Podcast 56 (ReDoS)" (http://www.owasp.org/index.php/Podcast_56). Retrieved 2010-04-02.

External links

- Examples of ReDoS in open source applications:
  - ReDoS in DataVault (http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2009-3277)
  - ReDoS in EntLib (http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2009-3275)
  - ReDoS in NASD CORE.NET Terelik (http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2009-3276)
  - ReDoS in .NET Framework (http://blog.malerisch.net/2015/09/net-mvc-redos-denial-of-service-vulnerability-cve-2015-2526.html)
  - ReDoS in an NPM package for AWS Lambda serverless applications (https://www.puresec.io/blog/redos-vulnerability-in-aws-lambda-multipart-parser-node-package)
- Some benchmarks for ReDoS
  - Achim Hoffman (2010). "ReDoS - benchmark for regular expression DoS in JavaScript (https://github.com/EnDe/ReDoS/)". Retrieved 2010-04-19.
  - Richard M. Smith (2010). "Regular expression denial of service (ReDoS) attack test results (http://www.computerbytesman.com/redos)". Retrieved 2010-04-19.
- Paper on implementing regular expressions not vulnerable to certain classes of ReDoS
  - Russ Cox (2007). "Regular Expression Matching Can Be Simple And Fast (http://swtch.com/~rsc/regexp/regexp1.html)". Retrieved 2011-04-20.

- Tools for detecting ReDoS vulnerabilities
  - H. Thielecke, A. Rathnayake (2013). "Regular expression denial of service (ReDoS) static analysis (http://www.cs.bham.ac.uk/~hxt/research/rxxr.shtml)". Retrieved 2013-05-30.
  - B. van der Merwe (2016). "The regular expression static analysis (ReDos) project page (http://www.cs.sun.ac.za/~abvdm/regex.html)". Retrieved 2016-08-12.
