FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Java vulnerability to regular expression denial of service
Student:	Andrey Olos
Supervisor:	Ing. Ondřej Guth, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Security and Information technology
Department:	Department of Computer Systems
Validity:	until the end of summer semester 2023/2024

Instructions

- Study the regular expression denial-of-service (ReDoS) attack.
- Discover vulnerable regexes and input strings that are susceptible to ReDoS attacks.
- Focus on java.util.regex implementation in a recent development version of OpenJDK [1].
- Provide thorough experimental evaluation for your findings.

[1] https://github.com/openjdk/jdk

Electronically approved by prof. Ing. Pavel Tvrdík, CSc. on 13 February 2023 in Prague.

Bachelor's thesis

JAVA VULNERABILITY TO REGULAR EXPRESSION DENIAL OF SERVICE

Andrey Olos

Faculty of Information Technology
Information Security
Supervisor: Ing. Ondřej Guth, Ph.D.
June 29, 2023

Contents

List of Figures

List of Tables

List of Algorithms

I would like to thank my supervisor Ing. Ondrej Guth Ph.D. for supporting, guiding, providing feedback, and giving useful advice. I am deeply grateful for your mentorship. I also extend my thanks to my lovely wife for her constant supporting, helping, and understanding. Lastly I would like to thank my family, my mother, father, and two sister, who always encouraged me to keep going.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on June 29, 2023 .....

Abstract

In this project, we study vulnerable regular expression inputs in the Javas platform implementation, also known as OpenJDK. These inputs can cause a Regular expression Denial of Service attack on the system which could make a targeted system unavailable to actual users. OpenJDK is considered protected against ReDoS attack, especially in versions 9 and higher where optimization of the regular-expression matching algorithms was added. But even with this improvement, we will show that there are still vulnerable regex inputs, which can cause exponential run-time problems and potentially cause a denial of service, even in the latest version of OpenJDK. In this thesis, you will see which optimizations were added in version 9, how these improvements affect OpenJDK today, and what these optimizations did not manage to cover. For each of the vulnerable regex inputs you will also be given a flow chart that shows the process of the matching algorithms, why the exponential run-time problems occur, and possible solutions.

Keywords ReDoS, Regular expression, Denial of Service, OpenJDK, Java, vulnerability

Abstrakt

V tomto projektu studujeme zranitelné vstupy regulárních výrazů v implementaci platformy Java, známé také jako OpenJDK. Tyto vstupy mohou způsobit útok regulárního výrazu Denial of Service na systém, který by mohl způsobit, že cílový systém nebude dostupný skutečným uživatelům. OpenJDK je považováno za chráněné proti ReDoS útoku, zejména ve verzích 9 a vyšších, kde byla přidána optimalizace algoritmů pro párování regulárních výrazů. Ale i s tímto vylepšením ukážeme, že stále existují zranitelné vstupy regulárních výrazů, které mohou způsobit exponenciální problémy za běhu a potenciálně způsobit odmítnutí služby, a to i v nejnovější verzi OpenJDK. V této práci uvidíte, které optimalizace byly přidány ve verzi 9, jak tato vylepšení ovlivňují OpenJDK dnes a co tyto optimalizace nestihly pokrýt. Pro každý ze zranitelných vstupů regulárních výrazů dostanete také vývojový diagram, který ukazuje proces párovacích algoritmů, proč dochází k exponenciálním problémům za běhu a možná řešení.

Klíčová slova ReDoS utok, Regularní výraz, OpenJDK, Java, bezpečnost, zranitelnost

List of abbreviations

DFA	Deterministic Finite Automaton
FA	Finite Automaton
NFA	Nondeterministic Finite Automaton
DOS	Denial of Service
ReDos	Regular expression Denial of service
Regex	Regular expression
JDK	Java Development Kit
JVM	Java Virtula Machine

Introduction

A regular expression (Regex) is a series of different characters that represent a pattern, these patterns are used by the regular expression engine to match input strings to the given patterns. In the world of IT this process is crucial and is used for a multitude of different tasks. They can be seen in the building of search engines, databases, in allowing input checking in applications, and are used in some of the most well-known Unix system commands like awk, sed, and grep. Nearly every publicly used programming language has the ability or an additional library that allows you to use Regex within it. Regex isn't just used in code, it is also a known tool in language theory that allows you to describe regular language.

As you can see, regular expressions are a diverse tool that can be used all around and in a lot of different contexts. For this exact reason, most Java systems hold some Regex within them, be it to help verify inputs or to check that a new password does not contain the user's email. Although this tool is helpful and should be used, there are still security vulnerabilities that can occur within it. One of the most dangerous vulnerabilities that can arise from regex is a Denial of Service (DoS). DoS, in computing, is a cyber attack that causes a system or network of systems to become unavailable to its intended users for a period of time by overloading their available resources. These attacks can cost companies thousands of dollars and seriously affect users who need access to the given network.

A regular expression DoS, or sometimes referred to as ReDoS, is the same type of attack, but in this case the attacker does not need a lot of resources, they just need to exploit a weak regular expression pattern. ReDoS attacks are algorithmic complexity attacks caused by an exploited regular expression triggering a large amount of backtracking. This causes the affected system to expand a large amount of resources on the request, which can cause a DOS.

Today, the Java programming language is used mostly for web application development. In the article *"Static Detection of DoS Vulnerabilities in Programs that Use Regular Expressions"* [1] an experimental evaluation was done, this evaluation found that approximately 20% of these Java web applications use vulnerable regular expressions within them. They found this by analyzing Java programs from GitHub repositories for their likelihood of a ReDoS attack. This shows us that ReDoS attacks are not just theoretical attacks, but actual ones that can be found in many of today's applications.

A good example of a real-life ReDoS attack occurred in 2016, where a successful attack was executed against the stackstatus.net server [2], this server is the StackOverflows status page where users can stay updated on incidence and site maintenance. This attack took down the whole of StackStatus and StackOverflow for 34 minutes. The fall was caused by a regular expression that was meant to trim the Unicode of white spaces from the beginning and end. The Regex pattern was: `^[\s\u200c]+|[\s\u200c]+$`, which caused an issue when an input of 20,000 white spaces was entered due to the backtracking that took place. The input was therefore checked by the engine 199,990,000 times which, though not exponential, was enough to overload the

server. Because this was done to the home page which was also used by the load-balancer for the health-check, the entire site became unavailable. Although it was only down for 34 minutes, that can be enough to seriously harm certain companies and sites.

In this project, we will look at a portion of vulnerable regular expressions patterns in the OpenJDK implementation [3], these patterns can be used to launch ReDoS attacks on a given system. In general, OpenJDK has added optimizations to its regex engine to reduce its chances of causing a ReDoS attack, especially in versions 9 [4]. But even with this improvement, we will show that there are still vulnerable regex patterns which can cause exponential run-time problems, catastrophic backtracking, and potentially cause a denial of service, even in the latest version of OpenJDK [3].

In this paper, you will see which optimizations were added in version 9 and how these improvements affect OpenJDK today. In addition, we will look at the regex patterns that have remained vulnerable within OpenJDK even after the version 9 optimization and will analyze the inner causes of their resulted exponential run-time. Each of our vulnerable patterns will be put through a thorough evaluation which will produce a graph of their run-time, and their object diagram.

The information we gather will hopefully help future developers understand the danger of specific groups of regex patterns and will help them avoid their use or potentially improve the underlying engine as a whole. At the end of this thesis, we will also present a few possible solutions to help mitigate the possibility of ReDoS attacks to Java developed applications.

Goal

In this thesis, we have four main goals that we will conquer. Our first being the discovery of vulnerable patterns and input text, which can cause exponential run-time in the system and potential cause denial of service. Our second task will be to investigate the OpenJDK implementation [3] to better understand the root cause of the vulnerabilities we have found. The third goal, you will see is to provide experimental evaluation and compare the run-time between other vulnerable patterns and different inputs. Our fourth and final goal will be to present theoretical possible solutions to prevent exponential run time based on investigations of the OpenJDK implementation.

Chapter 1
Preliminaries

In this chapter of our thesis, we aim to establish a clear understanding of the terminology used throughout our research. It is crucial to define and explain the key terms and concepts that we use to ensure accurate and consistent communication of ideas.

By providing a comprehensive glossary of terms, we enable readers to navigate our thesis with ease and clarity. This chapter serves as a reference point and helps to ensure that all readers share a common understanding when it comes to our terminology.

In addition, due to the importance of aligning our definitions with established and known industry standards, we have taken into account trusted sources and widely accepted definitions to ensure accuracy and consistency in our terminology usage.

1.1 Formal languages, classification, and automata

In this subchapter, we define important terminology used through our research. Definitions were taken from [5]

Formal languages

▶ **Definition 1.1** (**Alphabet**). *(denoted by Σ) is a finite set whose elements are called symbols.*

▶ **Example 1.2.** {1,2,3}, {a,b,c,d}

▶ **Definition 1.3** (**String**). *(word an alphabet) is a finite sequence of symbols from that alphabet.*

- ε - *the empty string, i.e., the empty sequence of symbols*
- $\Sigma*$ - *the set of all strings (including the empty one) over the alphabet* Σ
- $\Sigma+$ - *the set of all non-empty strings over the alphabet* Σ.

▶ **Example 1.4.** Let define $\Sigma = \{a, b, +\}$ Then $a, b, aa, ab+, \varepsilon$ are strings over the alphabet Σ.

▶ **Definition 1.5** (**Formal language** L). *is any subset of the set of all strings over* Σ

- ***enumeration notation:*** $L_1 = \{\epsilon\}$, $L_2 = \{1, 2\}$, $L_3 = \{0, 00, 01, 10, 11\}$
- ***set-builder notation:*** $L_4 = \{0^n 1^n : n \in \mathbb{N}\}$

Classification of languages (the Chomsky hierarchy)

▶ **Definition 1.6** (**Regular languages**). *represent the simplest set of formal languages. A formal language is labeled regular if and only if it can be generated by a regular grammar. For each regular language, it holds that it can be:*

- *recognized by a (non)deterministic finite automaton (DFA)*
- *described with a regular expression*

▶ **Definition 1.7** (**Context-free languages**). *represent another type of formal languages. A formal language is labeled context-free if and only if it can be generated by a context-free grammar. For each context-free language, is also holds that it can be:*

- *recognized by a nondeterministic push-down automaton*

▶ **Definition 1.8** (**Context-sensitive languages**). *represent another type of formal languages. A formal language is labeled context-sensitive if and only if it can be generated by a context-sensitive grammar. For each context-sensitive language, it also holds that it can be:*

- *recognized by a nondeterministic linear bounded automaton (i.e. linear bounded Turing machine)*
- *generated by a non-contracting grammar*

▶ **Definition 1.9** (**Recursively enumerable languages**). *represent the fourth type of formal language. A formal language is labeled recursively enumerable if and only if it can be generated by an unrestricted grammar. For each recursively enumerable language, it also holds that it can be:*

- *recognized by a (non)deterministic Turing machine*

Automata

▶ **Definition 1.10** (**Finite automat**). *Formally, a finite automaton is defined as a quintuple* $M = (Q, \Sigma, \delta, q_0, F)$ *where*

- Q *is a finite non-empty set of states*
- Σ *is a finite set of symbols called the input alphabet*
- δ *is the transition function*
- $q_0 \in Q$ *is the finite (start) state*
- $F \subseteq Q$ *is a set of final states*

▶ **Definition 1.11** (**Deterministic finite automaton (DFA)**). *is a finite automaton that always knows how to continue during the computation (multiple choices are not available). Therefore, we define its transition function as follows:*

- δ *is mapping from the set* $Q \times \Sigma$ *to the set of states* Q *(i.e.,* $Q \times \Sigma) \longrightarrow Q$

▶ **Definition 1.12** (**Nondeterministic finite automaton (NFA)**). *is a finite automaton that at some point of computation has multiple choices on how to continue. Therefore, we define its transition function as follows:*

- δ *is a mapping from the set* $Q \times \Sigma$ *to the power set (set of all subsets) of* Q *(i.e.,* $\delta : Q \times \Sigma \longrightarrow P(Q)$*)*

1.2 Regex syntax and semantics in OpenJDK

▶ **Definition 1.13.** *[**Regular expression**] is a sequence of characters that specifies a match pattern in text. Originally introduced by [6] to describe regular languages as described in definition 1.6.*

▶ **Definition 1.14.** *[**OpenJDK**] is a free and open source implementation of the Java platform, Standard edition (Java SE).[7]*

A regular expression, commonly referred to as regex, is a powerful pattern-matching syntax that is used to search, manipulate, and validate text. It does all this by allowing us to build flexible but complex search patterns through the combination of characters, meta-characters, and special sequences.

Regular expressions can be found nearly anywhere within various programming languages, text editors, command-line tools, and more. The implementation of regex in Java is located in the *java.util.regex* package, which provides classes for working with regular expressions in Java code. In this package are two main classes, which implement the whole regex engine:

- **Pattern** - defines a regular expression pattern and build object tree for matching algorithm.
- **Matcher** - applies a pattern to an input string and provides methods to perform various operations on the matching string.

The syntax of regular expressions in Java is based on Perl-style regular expressions [8]. OpenJDK regular expression patterns support a wide range of character classes, quantifiers, and backreferences features [9]:

▶ **Definition 1.15** (**Character classes**). *defines a set of characters that can be matched in the input string. Examples of character classes:*

- $[a, b, c]$ - *match any characters in the set of* $\{a, b, c\}$
- $[\wedge a, b, c]$ - *match any character, which is not in the set of* $\{a, b, c\}$
- $\backslash d$ - *match any digit character*

▶ **Example 1.16.** Pattern $[xyz]$ match strings: $x, xzz, xxyy, xxyzz$, etc. $L = \{w : w \in \{x, y, z\}\}$

▶ **Definition 1.17** (**Quantifiers**). *gives the ability to define the number of occurrences to match a pattern. Examples of quantifiers in Java:*

- $+$ - *match one or mare instances of the preceding pattern*
- $*$ - *match zero or more instances of the preceding pattern*
- ? - *match zero or one instance of the preceding pattern*
- $\{n, m\}$ ***where*** $n, m \in \mathbb{N}$ - *define range of repetitions of the preceding pattern*

▶ **Example 1.18.** Pattern $xxa+$ match strings: $xxa, xxaa, xxaaa$, etc. $L = \{xxaw : w \in \{a\}*\}$

▶ **Definition 1.19** (**Capturing groups**). *are a way to treat multiple characters as a single unit. They are created by placing the characters to be grouped within a set of parentheses. [10]*

▶ **Example 1.20.** In pattern $(a+)b$ by parentheses we define a subpattern - $a+$.

▶ **Example 1.21.** In pattern $(a+)bbb(cd+)$ by parentheses, we define two subpatterns. The first is $a+$ and the second is $cd+$.

▶ **Definition 1.22** (**Backreference**). *is a feature that allows for reference to a previously captured group within the pattern itself. It allows the engine to match the same group or sequence of characters that were previously matched within the capture group. This gives us the ability to identify repetitive patterns.*

A backreference is usually represented within our pattern by a special syntax; in the case of Java it is shown by two backslashes and a number (e.g., "\\1", "\\2", etc.). The number shown represents the number of captured groups; for example, \1 refers to the first capturing group in our pattern, while \2 refers to the second capturing group. When a backreference is used, the regex engine will try to find a match within our input text for the same substring that was in the desired reference group.

▶ **Example 1.23.** To be able to define context-sensitive language $L = \{a^n ba^n | n \geq 1\}$, within a regex pattern, it will be expressed as $(a+)b\backslash 1$, where \1 points to the capture group $(a+)$.

▶ **Example 1.24.** To define $L = \{(ab)^{2n} c^m dc^m | n \geq 1, m \geq 0\}$ within a regex pattern, it will be expressed as $(ab) + \backslash 1(c*)d\backslash 2$.

1.3 Backtracking in NFA-based regex implementation

▶ **Definition 1.25** (**Backtracking matching algorithm**). *is an algorithmic technique that makes use of an input-directed depth-first search on an NFA regex engine. [11]*

Catastrophic Backtracking in regex is when a regular expression engine spends an excessive amount of time trying to match a string that does not match the pattern. This process can cause exponential run-time, stack overflow, and a variety of other problems.

▶ **Example 1.26.** An example of the way backtracking works within OpenJDK: For the regular expression $(a+)+!$ with input *aaaa*, in the case when backtracking is enabled, it will get through and check the following way:

Match $(a+)+!$ to input *aaaa*		
Match Attempt Number	Attempt to Match $(a+)+$	Attempt to Match Character '!'
1	*aaaa*	Not Found
2	*aaa* — *a*	Not Found
3	*aa* — *aa*	Not Found
4	*a* — *aaa*	Not Found
5	*a* — *a* — *aa*	Not Found
6	*a* — *aa* — *a*	Not Found
7	*aa* — *a* — *a*	Not Found
8	*a* — *a* — *a* — *a*	Not Found

Table 1.1 Example of backtracking for non-matching input

This table was created as part of the analysis that will be presented in Chapter 3.

▶ **Definition 1.27** (**NFA based regex implementation**). *One of the methods to implement the regex engine. This method constructs NFA, as described in definition 1.12, for the given regular expression and by using recursive backtracking to find the match for the given string. The time complexity in the NFA native implementation is $O(n^2)$, where n is the length of the string [12]*

▶ **Definition 1.28** (**DFA based regex implementation**). *Another method of implementing the regex engine is to construct DFA, as described in definition 1.12, automata for the given regular expression and use it to match the string. The time complexity guarantee $O(n)$, where n is the length of the string, but we might end up having 2^n states due to the state explosion problem [12]*

The Java regex engine is a traditional NFA-based implementation [9], which uses recursive calls to implement its backtracking. This kind of implementation uses backtracking as a way to help match the desired input to the provided pattern, and in cases where a match is not found, this method can result in catastrophic backtracking.

As defined in definition 1.2, Java supports features such as backreferences and capturing groups. Backreferences help us to define context-free languages , defined in definition 1.7, (see example in table 1.23), which cannot be recognized by the deterministic final automaton (DFA). This is the main reason why Java does not use the DFA-based implementation.

1.4 OpenJDK Versioning

OpenJDK has in the past implemented changes to help mitigate the negative effects of backtracking, with the most impactful change being made in version 9 [4] (released on September 21st 2017). With versions 8 [13] and before, backtracking was always enabled with no way of turning it off. From version 9 and beyond, this is no longer the case; a new variable was introduced to the regex engine that indicates to the algorithm whether backtracking is enabled. This feature helped reduce the amount of catastrophic backtracking that occurred within the regex engine and helped reduce the run-time for specific patterns. The criteria for backtracking being disabled is not fully clear, but previously vulnerable patterns such as $(a+)+$ that caused experiential run-time in version 8 in OpenJDK are no longer vulnerable in versions 9 - 20. This will be better looked at in our second chapter in figure 2.5 and in our third chapter in figure 3.2.

Although backtracking has been disabled for a small set of vulnerable patterns in versions 9 through 20, it is still enabled for more complex vulnerable patterns. These patterns will be thoroughly explored in the next chapters.

1.5 Regular expression Denial of Service (ReDoS)

▶ **Definition 1.29** (**ReDoS**). *is an algorithmic complexity attack that produces a denial-of-service by providing a regular expression and/or an input that takes a long time to evaluate. [14]*

As mentioned before, OpenJDK's regex engine is based on an NFA implementation, and this implementation allows for backtracking during its computation. When the engine is fed a pattern, it will build a straightforward nondeterministic automaton. From there the algorithm will take any given input string and try to see if it can be considered a match by going forward and checking all possible paths, or as many as it takes to find a viable one, this is done using backtracking. In a worst-case scenario, all where all of the paths are tried and fail, which should result in a negative result, but due to the run-time and resource usage that this problem can take, we can be left with our systems being unable to respond to other requests or even a stack overflow. This is defined as a ReDos attack, when a system experiences a denial of service due to a regular expression. A good example of real-life regex patterns that were vulnerable to ReDoS attacks before OpenJDK 9 can be found in[15].

Chapter 2

Vulnerable regex patterns in OpenJDK

This chapter investigates vulnerable regex patterns in OpenJDK and provides an experimental evaluation of their impact. Regex is a powerful tool used for pattern matching in software development, but certain incorrect patterns that can be created in regex can introduce security vulnerabilities. In this research, we discover and analyze vulnerable regex patterns with different input texts in the OpenJDK project. We will categorize them based on their security risks and assess their impact through a series of experiments. Our findings shed light on the importance of secure regex usage and provide insight for developers to avoid similar vulnerabilities in their own project.

2.1 Introduction

The regex engine in Java is a powerful tool for validating, processing, and filtering input texts, but with great power in IT comes even greater vulnerabilities. When patterns are not carefully crafted and properly used, they can introduce vulnerabilities that attackers can exploit. These vulnerabilities can range from denial-of-service attacks due to catastrophic backtracking to information disclosure and remote code execution.

This chapter will provide examples of these types of patterns which are vulnerable in the new OpenJDK 20 (released 2023). It will provide background information on each pattern and the qualities that make them a potential threat. From there we will be able to look at the run-time analysis of each example to show the true depth of the issue.

The remainder of this chapter is organized as follows. Subchapter two describes our methodology for identifying and categorizing vulnerable regex patterns in OpenJDK. Subchapter three presents the vulnerable regex patterns that were found. Subchapter four will present the experimental setup and evaluation of the identified vulnerabilities. Finally, subpattern five concludes the whole chapter by reviewing our findings and presenting the conclusion of this chapter.

2.2 Methodology for identifying vulnerable regex pattern

To identify vulnerable regex patterns in OpenJDK, we followed a systematic methodology that contains five steps.

The first step was to identify our vulnerable criteria. When doing this, we first must look at the most well-known vulnerabilities within the regex engine. This includes catastrophic back-

tracking and exponential time complexity; this will be our vulnerable criterion. These criteria will determine which patterns we are looking for and which we will exclude for this analysis.

From there we can move to our second step, finding examples of patterns that fit these criteria by using research papers, manual testing, forums such as StackOverflow, OpenJDK github bugs, and OpenJDKs' own bug ticketing system. By utilizing these and other sources, we can easily see and collect the multitude of different patterns that have caused these problems for real-life users.

Once we have a large enough collection, we can start the third step, categorizing. Here, we can break each pattern down and create the most basic form of the pattern that still fits our vulnerable criteria. This allows us to remove patterns that were found that are caused by the same root issue and categorize each remaining regular expression. This categorization allows us to analyze and evaluate the severity of each pattern and prioritize further investigation.

For our fourth step, we take the remaining regex patterns that meet our vulnerable criteria and pose potential risks and now conduct experiments to evaluate their impact. These experiments are shown in section 2.4.4 and include runtime and performance-based tests that compare the results when fed different inputs.

Our fifth and last step is documenting and reporting. Throughout the process, we aim to identify regex patterns, their categorization, vulnerability analysis, and experimental results; This documentation serves as a basis for further analysis, discussion, and reporting of our findings.

By following this methodology, our aim is to identify and better analyze the vulnerabilities within OpenJDK pertaining to the regex engine. This approach best allows us to uncover these potential security risks and provide the best and most thorough results regarding them.

2.3 Vulnerable patterns in OpenJDK

In this subchapter we will show examples of vulnerable patterns. It is important to note that in our examples we use the letter "a" for simplicity, though "a" can be substituted by any substring or special symbol.

In addition, in most of our examples, we will use a Kleene plus which can be easily replaced by a Kleene star and still be considered a vulnerable pattern.

2.3.1 Pattern $((a+)+)+$ with input $a^n!$

This type of vulnerable patterns is one of the most famous security issues in the regex engine and can be found in different sources such as the OWASP security documentation [15], in the about detection of ReDoS [1] article, and in the software engineering forums such as StackOverflow [16]. This kind of pattern can also be found in the official OpenJDK bug system [11], a system that allows users to report problems and once authenticated become open bugs that OpenJDK is aware of that need to be fixed.

The pattern $((a+)+)+$ matches strings that contain nested sequences of one or more occurrences of the letter "a". The regex engine of OpenJDK matches huge inputs of "a"s quickly. The problem arises when the input text does not fit the pattern. For example, if we add to the end of an input string of "a"s the symbol "!". Here, the algorithm starts backtracking; this causes exponential run-time of the application.

OpenJDK version 8 had exponential backtracking with the pattern $(a+)+$, but starting with version 9 this issue was resolved, as mentioned in section 1.4 and can be seen in the figure 2.5. It did so by adding optimization to prevent exponential backtracking in this type of pattern (which will be explained in section 3.4). Unfortunately, by adding an additional capturing group with quantifier - "$()+$" we can still cause exponential backtracking. When looking closer at the issue, we can see that adding more and more capturing groups to the pattern, the execution time will become longer and longer. A good example can be seen later in the figure 2.1, a regular

expression pattern such as ((((a+)+)+)+)+ fed a small input such as a^{16}! can take more than a minute on a regular PC in OpenJDK 20 to return the negative result, as can be seen in figure 2.1.

2.3.2 Pattern $(a+)\{1,100\}$ and $(a\{1,100\})\{1,100\}$ with input a^n!

Sourced from the research document [17], where this kind of pattern was described as performance problematic in NFA backtracking algorithm. These patterns are similar to the previous pattern, $(a+)+$, the difference between them is that we define the repetitions manually. Although unlike in pattern $a+$, in this pattern our character a can repeat only within the range given, in this case 1 to 100.

Despite the many similarities between these patterns, the interesting difference between them is that the pattern $(a+)+$ is not vulnerable in Java since version 9, but the patterns $(a\{1,100\})\{1,100\}$ and $(a+)\{1,100\}$ are vulnerable even in version 20.

2.3.3 Pattern $(a|a)\{1,100\}$ with input a^n!

First was sourced from [18], where it explains about the regex implementation of Java version 6 and the vulnerable pattern $(a|a)*$ with non-matchable input. After personally manually testing this pattern in the new version of OpenJDK 20, the matching run time was quick and it no longer looks vulnerable, but when switching from the quantifier plus to the manual repetition setting quantifier, the exponential run time problem returns, as can be seen in figure 2.3.

The pattern $(a|a)\{1,100\}$ is similar to $(a|a)+$, and the situation is similar to the issue we describe in section 2.3.2. The suspicious difference between the patterns is that the $(a|a)+$ pattern is not vulnerable, since Java version 9 and the $(a|a)\{1,100\}$ pattern are vulnerable even in the new Java version released 2023.

2.3.4 Pattern $(a+)+\backslash 1$ with input a^n!

This pattern was discovered by me during our testing process by personally manually testing different patterns combining capturing groups and quantifiers. After adding a backreference to the capturing group at the end of the nonvulnerable pattern "$(a+)+$", another vulnerable pattern was found: "$(a+)+\backslash 1$".

This pattern accepts the same inputs as our nonvulnerable pattern $(a+)+$ but due to the backreference added at the end the time complexity of our run time increases drastically.

2.4 Experimental evaluation

In this section we will go into the fourth step of our process of discovering and analyzing vulnerable regex patterns. More specifically, our vulnerable patterns that we looked at in section 2.3. We will show the experimental evaluation that was conducted to assess the impact of each found vulnerable regex pattern and the different inputs we used to achieve these results. The purpose of this evaluation is to measure the performance and potential security implications of these vulnerable patterns. This will give us a better understanding of the outcome these patterns can have when used within any code.

2.4.1 Controlled test environment setup

To obtain the best results with the least impact, we created an executor service that will run our input strings, one by one, within the vulnerable pattern using a single thread. Using the VisualVM monitoring system, we can monitor the running thread and get the exact run-time of the matching algorithm. This monitoring application will also let us be sure that our testing environment is clean and that no other threads or processes are running that might skew our results. In addition, to reduce the enslavement of a third-party processes in our experiment, we will turn off all possible applications and processes that are running in operation systems, like antivirus, IDE and so on.

It is important to note that for measurements of less than 10 milliseconds VisualVM was not used as it cannot measure that small amount of time. For measurements below 10 milliseconds, the method $System.nanoTime()$ was used, which measured the time the process started and the time it finished and deduced the start from the end. The difference

For our testing, we will use only OpenJDK commands, such as the *javac* command to compile and the *java* command for running tests. We run our test with one JVM argument that allows the RMI (Remote Method Invocation) localhost server for VisualVM to monitor CPU time.

2.4.1.1 Platform specification

- **Operation system:** Windows 10 Enterprise
- **Processor:** 12th Gen Intel® Core™ i7-12800H 2.40 GHz
- **Architecture:** x64-based processor
- **Memory size (RAM):** 64.0 GB

2.4.1.2 OpenJDK 20 and Java Virtual Machine (JVM) specification

- **OpenJDK version and build:** 20.0.1 2023-04-18 (build 20.0.1+9-29)
- **JVM vendor:** Oracle Corporation
- **JVM version:** 20.0.1+9-29
- **JVM configuration:** -Djava.rmi.server.hostname=localhost

2.4.1.3 OpenJDK 8 and Java Virtual Machine (JVM) specification

- **OpenJDK version and build:** 1.8.0_43 (build 25.40-b25)
- **JVM vendor:** Oracle Corporation
- **JVM version:** 1.8.0_43-b03
- **JVM configuration:** -Djava.rmi.server.hostname=localhost

2.4.1.4 Monitoring tool

- **VisualVM version:** 2.1.6

2.4.2 Selection of vulnerable regex patterns

To understand the difference between these results compared to safe regex patterns, it is important to note that running the same input on a safe version of one of these patterns, $(a+)+$, the run-time was so low that it resulted in less than one milliseconds on our test monitoring system.

Experiment 1: We will run three tests in OpenJDK 20 with patterns that include Kleene plus: $((a+)+)+$, $(((a+)+)+)+$, and $((((a+)+)+)+)+$.

Experiment 2: We will run three tests in OpenJDK 20 with patterns that include Kleene stars: $((a*)*)*$, $(((a*)*)*)*$ and $((((a*)*)*)*)*$.

Experiment 3: We will run three tests in OpenJDK 20 with patterns that include manual repetitions: $(a+)\{1,100\}$, $(a\{1,100\})\{1,100\}$ and $(a|a)\{1,100\}$.

Experiment 4: We will run two tests in OpenJDK 20 with patterns that include a backreference: $(a+) + \backslash 1$ and $(a+)\backslash 1$.

Experiment 5: We will run the pattern $(a+)+$ in OpenJDK 8 vs. the same pattern in OpenJDK 20

2.4.3 Input scenarios

To keep an even testing scenario among all our tests, we will use only one matching character: 'a'(index 97 in ASCII table). This means that all our patterns will correctly match any input string of any length, excluding the ones that contain a min and max, as they will only match if their length fits within the given range, which contains only the character a.

Within our testing, we are trying to cause catastrophic backtracking within the matching algorithms and exponential run-time. This can be done by attempting to match an input string to a pattern that does not match. For this reason, we will add an invalid character to the end of our input string in every test. Our invalid character will be – "!".

Our input strings will range from two characters long, one "a" character and one "!" character, to 31 characters long, 30 "a" characters, and one ! character.

Example of our input string: $a!$, $aa!$, $aaa!$...

2.4.4 Performance measurement

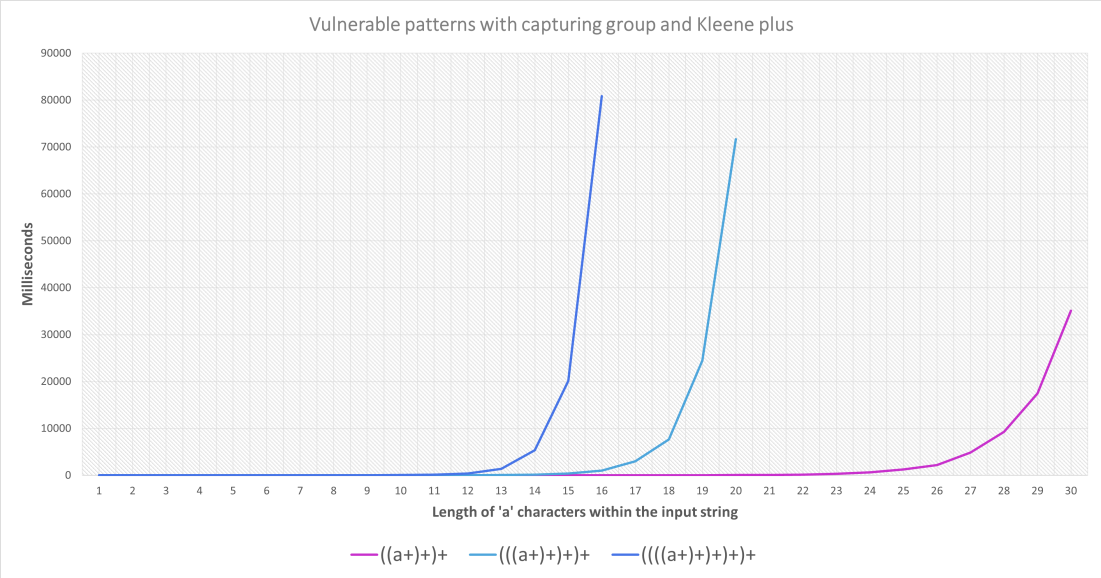

Figure 2.1 Graph results of patterns $((a+)+)+$, $(((a+)+)+)+$, and $((((a+)+)+)+)+$

This graph shows the results of the our first experiment mentioned in subsection 2.4.2. Here, we can see and compare the run-time of three regex patterns that hold the same acceptance criteria. Though these three patterns can be simplified to the same pattern, their runtime differs when given the same inputs. We can see that with every capturing groups added to the pattern, the run time increases expediently, with the input $a^{16}!$ taking 6.4 milliseconds for the pattern $((a+)+)+$, 991.9 milliseconds for the pattern $(((a+)+)+)+$, and 80828 milliseconds for the pattern $((((a+)+)+)+)+$.

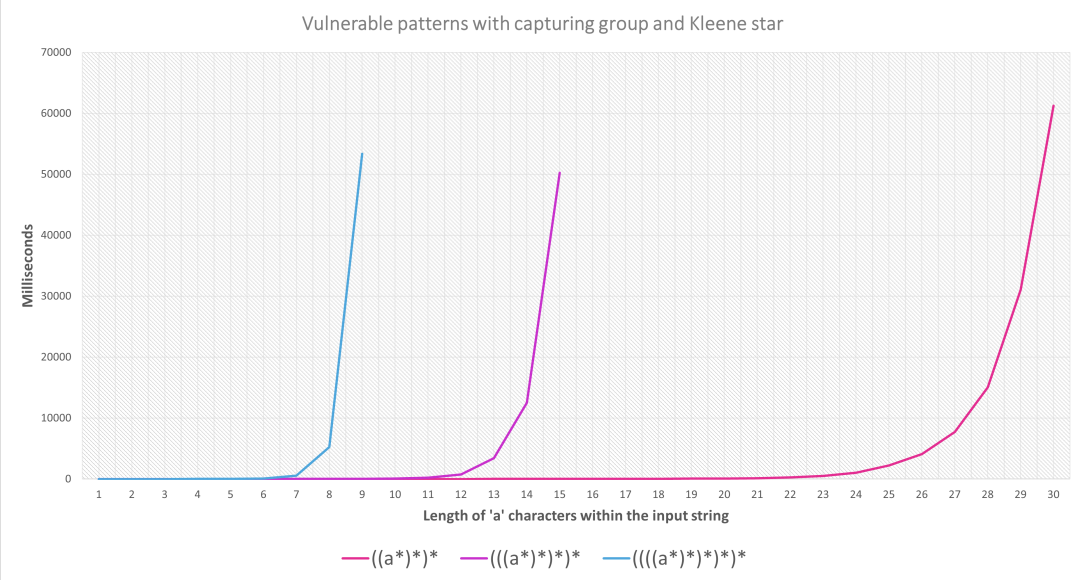

Figure 2.2 Graph results of patterns $((a*)*)*$, $(((a*)*)*)*$, and $((((a*)*)*)*)*$

This graph shows the results of our second experiment mentioned in subsection 2.4.2. Here, we can examine the run-time results for three patterned that use the Kleen star; the patterns also accept the same language criteria as one another. When comparing the three patterns to each other, we can intently tell a large run-time difference from each other. If we look at their run-time when given an input of $a^9!$ the pattern $((a*)*)*$ took 0.3603 milliseconds to run, the pattern $(((a*)*)*)*$ took 7.849501 milliseconds, while the pattern $((((a*)*)*)*)*$ took 5218 milliseconds.

Nearly identical patterns to these can be seen in the figure 2.2, all three patterns accept the nearly same inputs as those patterns, with the only difference being that they also except for an empty string, ε. But if we compare the run time of these two groups of patterns, we can see that they differ expediently. If we take the run-time of the patterns $((a+)+)+$ and $((a*)*)*$, we can see that patterns using the Kleen star have a much larger run-time; with an input of $a^{19}!$ taking 45.0002 milliseconds and an input of $a^{20}!$ taking 64.818 milliseconds with the pattern $((a+)+)+$, and the same inputs taking 53.2367 milliseconds and 86.5807 milliseconds for the pattern $((a*)*)*$.

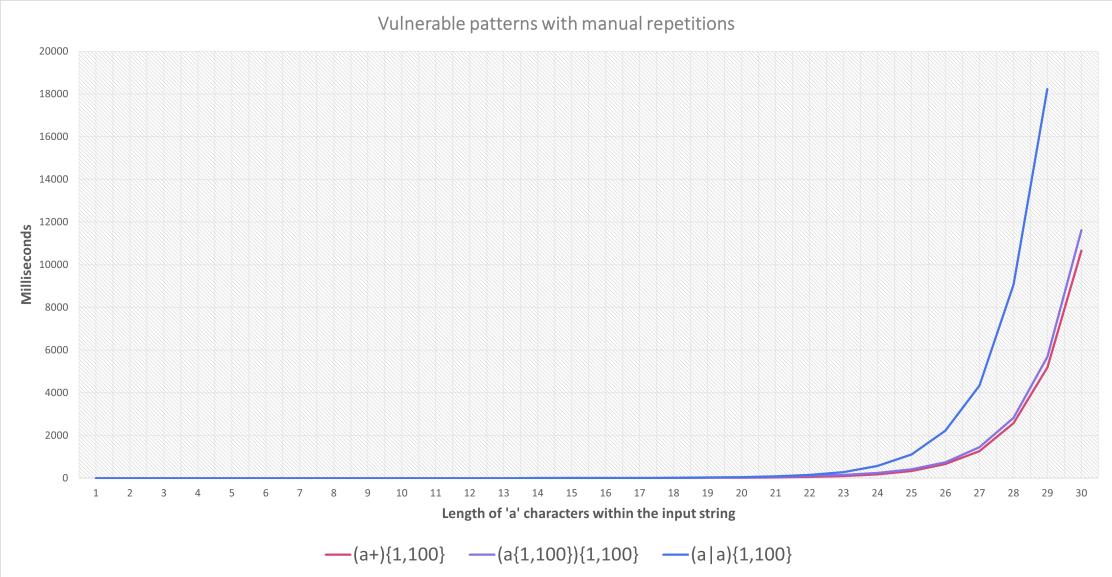

Figure 2.3 Graph results of patterns $(a+)\{1,100\}$, $(a\{1,100\})\{1,100\}$, and $(a|a)\{1,100\}$

Here we can see the graph representation of our results following the third experiment listed in subsection 2.4.2. Within this experiment we tested the run-time of our vulnerable patterns that use manual repetition within them. In this figure, we can see that all three patterns have quite a similar result with the pattern that uses the pipeline having a bit of a higher run-time out of the three. It is best seen when these patterns are given the input of $a^{29}!$; the pattern $(a+)\{1,100\}$ run time for this input is 5184.9429 milliseconds, the pattern $(a\{1,100\})\{1,100\}$ has the run time of 5690.746 milliseconds for this input, and the pattern $(a|a)\{1,100\}$ has the run time of 18225.0763 milliseconds for this input.

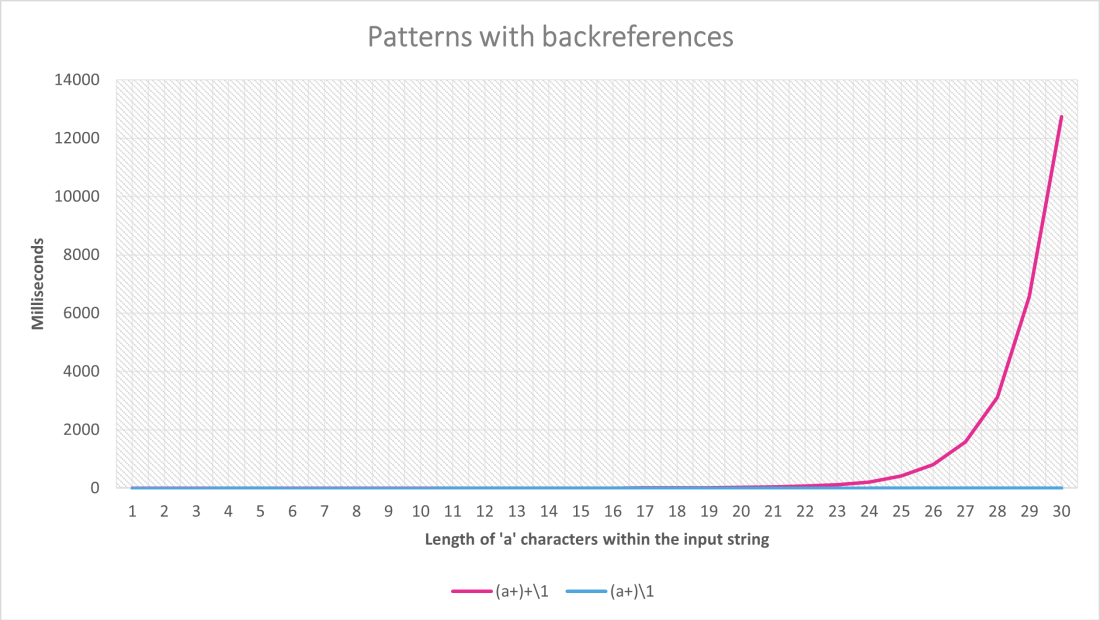

Figure 2.4 Graph results of the patterns $(a+)+\backslash 1$ and $(a+)\backslash 1$

In this graph, we show the results of our fourth experiment, as mentioned in subsection 2.4.2. As you can see from this graph, within this experiment, we have only tested one pattern, our reference pattern back, as mentioned in section 2.3.4, $(a+)+\backslash 1$. Here, we can see the exponential growth of this pattern when fed different input lengths.

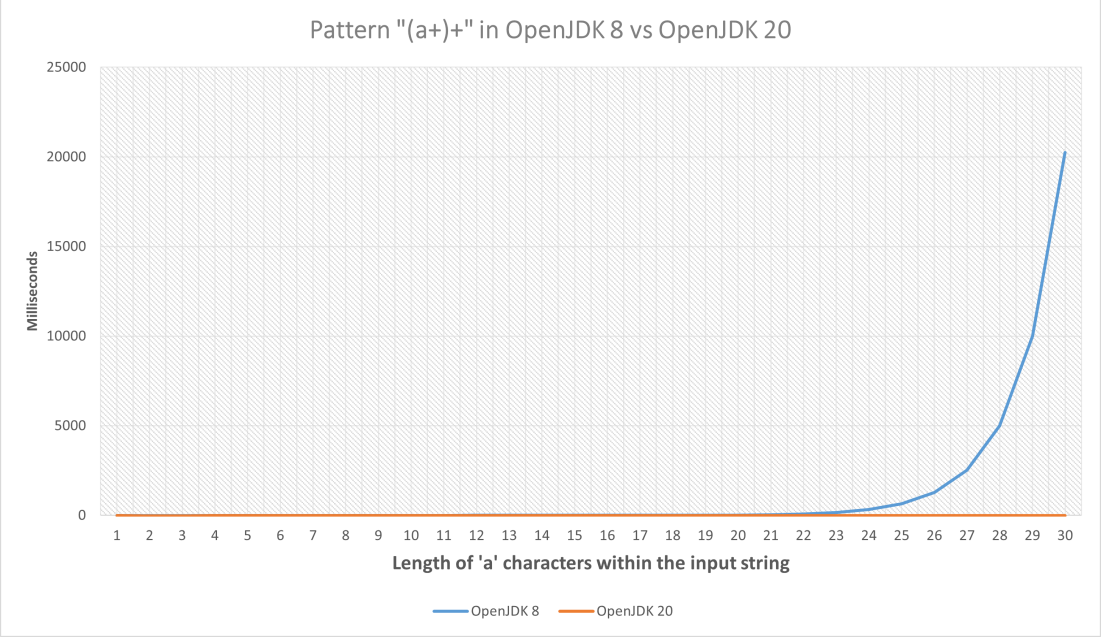

Figure 2.5 Graph results of the pattern $(a+)+$ in OpenJDK 8 vs OpenJDK 20

In this graph, we compare the matching run-time of pattern $(a+)+$ in different versions of OpenJDKs, for our fifth and final experiment as mentioned in section 2.4.2. From the results can

be seen the effect of the optimization mentioned in chapter 1.4 that was added after version 8 of OpenJDK. We can see that when the pattern is run in version 8 it exhibits the same experiential run-time problem as its sister patterns as can be seen in figure 2.1, unlike in version 20 of OpenJDK where this problem can no longer be seen when running this pattern.

2.5 Conclusion

What we can conclude from all the data we have collected during our investigation to identify vulnerable regex patterns is that even in the new versions of OpenJDK there are still potential security risks. We can see from all the data and evaluations that vulnerable patterns still exist within the regex engine. These patterns can cause serious problems such as exponential run-time problems.

Through our evaluation, we can identify security implications associated with the presence of capturing groups, repetitions, and backreferences within our regex pattern. These issues expose the system to potential denial-of-service attacks or other security vulnerabilities. Moreover, we found similar patterns like $(a+)+$ and $(a+)1,100$, which behave absolutely differently in terms of execution time even when fed the same inputs. It is important to remember that this difference in run-time happens only in version 9 and higher.

Now that we have established our vulnerable patterns, we can investigate the reason why they behave this way. In our next chapter we will go deeper into the implementation of the regex engine and will try to find the root cause of these problems.

Chapter 3

Analysing the causes of exponential run-time for vulnerable patterns in OpenJDK

This chapter aims to analyze the causes of the exponential run-time behavior observed in vulnerable patterns within the OpenJDK regex implementation. By understanding the underlying causes of this behavior, developers can understand the underlying reason for this behavior, devise strategies to mitigate its impact, and enhance the efficiency of regex processing in OpenJDK.

The following chapter is divided into five subchapters. The first will show a brief overhead of the implementation of the regex engine within OpenJDK. The second will show the construction of the object diagrams that were constructed for each pattern. The third will present the pseudo-code for important methods and provide a small description for each one. Our fourth subchapter will take all the information provided so far and give an analysis of the reason behind the exponential execution. Lastly, in our fifth section, we conclude our chapter.

3.1 OpenJDK regex engine implementation

The Java regex engine is fully located within the package - *java.util.regex*. Its core implementation is divided into two classes, which where briefly described in the definitions 1.2. These classes, *java.util.regex.Pattern*, and *java.util.regex.Matcher*, who's full description and a more in-depth analysis can be found in the research paper *"Analyzing Catastrophic Backtracking Behaviour in Practical Regular Expression Matching"*[18], are what we will be looking into to understand the algorithm behind the matching process.

We will first focus on the *Pattern* class; this class builds a tree of nodes, nodes being objects, based on the regex pattern we supplied to it. In the *Pattern* class, we can find the subclass *java.util.regex.Pattern.Node*, this subclass lets us add nodes in our tree. Nodes are objects that act like states, allowing us to move through them while comparing our characters in our input to the possible accepted characters that match our pattern. This subclass contains the method *match*; the *match* method is a fundamental component of the regex engine, responsible for determining whether a given input string matches a specified pattern. The subclass also has the reference pointer *next* that points to the next node in our tree.

With every character in our input we will slowly move along our constructed tree seeing one by one if it fits the given pattern. If the character is found to be a fit and is not the end of the input string, we will continue on to the next node. When a character is found to be not fitting,

the algorithm will return false.

The subclass *java.util.regex.pattern.Node* can be taken as a base class or parent class for all other node classes, and all child node classes should override the *match*() method and the *next* reference. The parent *Node* class is an accepting node, so the *match* method always returns true. A good example for a child node is *LastNode*, which is the final state, if we reach this node, our string is accepted and matches the input.

The object *Matcher* is created from a pattern by invoking the pattern's *matche* method.[19] This object is used like an open book and saves the result of a matching algorithm at any given time. The *Pattern* class only builds the tree but cannot remember the states in which we moved, so to remember the states, we need an object *Matcher*.

As described above, each node class should be overridden by the parent node method `match(Matcher matcher, int i , CharSequesnce seq)`, where *seq* is the input text, "*i*" is a current index of the input text itself and invoked *Matcher* object.

3.2 Constructed object diagrams of regex patterns in Pattern class

In this subchapter we will try to take an in-depth look at the classes *Pattern* and *Patcher*. We will illustrate the object diagram (node tree) that is created inside the class *Pattern.java* for each of the vulnerable and nonvulnerable patterns. We can do so by debugging our patterns when they are fed inputs and looking into the OpenJDK code to see what variables and classes they are using.

The process of doing so consists of using an *IntellijIDE* [20] to go step by step through the regex engine, each time with a different pattern set, and following its flow through the code. By doing so, we can see each variable and the value it holds at every step, which classes are entered, and what triggered each step. After observing and documenting the flow of each pattern, we can go into the code itself and investigate the way the engine was built and the logic behind it.

In the following diagrams, you will see boxes, these represent objects that were created in the code when the pattern was run. Within our objects there will be the variables with the value they held during the run. The path, an arrow, represents a dependency of another object and most of them are the next node.

Each of the following diagrams will have the object *Pattern* that holds within it our regex pattern. The *Pattern* also has a value named *matchRoot* that is a reference to a node tree that was built for the regex pattern and used by *Matcher* to match the input string. (see previous subchapter 3.1)

3.2.1 Diagrams of patterns $a+$, $(a+)+$, $((a+)+)+$

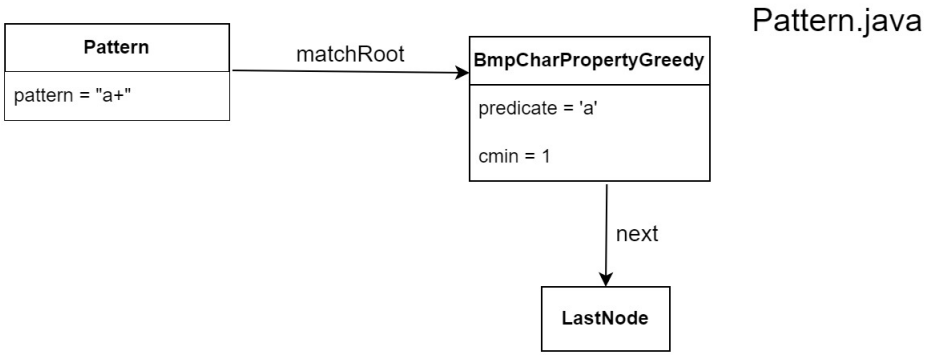

Figure 3.1 Constructed object diagram for non-vulnerable pattern $a+$

For pattern $a+$, within the OpenJDK regex engine, two nodes were created: *BmpCharProperty* and *LastNode*. The *BmpCharProperty* node has two values – *predicate*, which has the symbol that will be a match for our pattern a, and *cmin* whose value represents the minimum number of repetitions for the character a. *LastNode* is the final node and represents the last state in which an additional validation is performed. The additional validation checks to see if we have reached the end of our sequence.

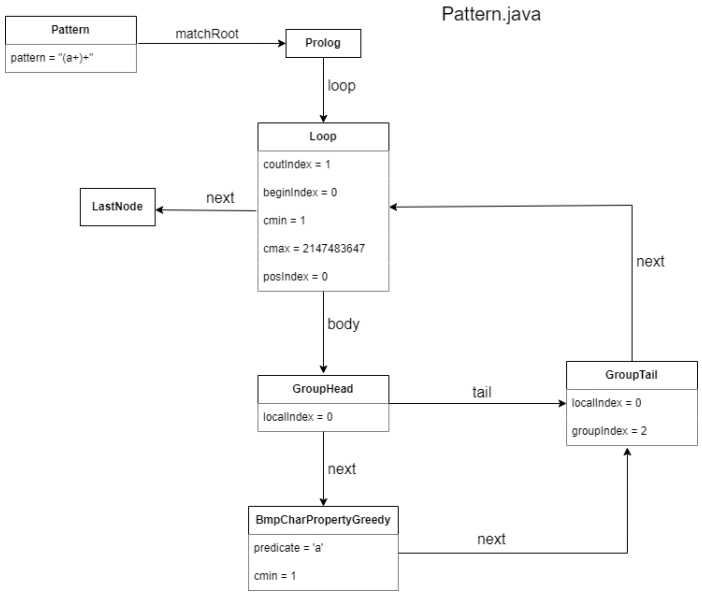

Figure 3.2 Constructed object diagram for non-vulnerable pattern $(a+)+$

For pattern $(a+)+$ an additional three objects, in addition to the two that were created for the previous pattern, were added - *Loop*, *GroupHead* and *GroupTail*.

The *Loop* object in our diagram represents the Kleen plus within our pattern that comes after the capture group. The variables *cmin* and *cmax* that are seen within the object show the minimum and maximum repetition that our capturing group is allowed, the maximum being 2147483647 as that is the largest numerical value the variable can hold. The value *posIndex* is set to 0, this value represents the algorithm if backtracking is activated or deactivated. Because

it is set to 0, we know that backtracking is disabled. There are other values such as *countIndex* and *beginIndex* which are tracking the capturing group. There is also the object *Prolog*, which is an initialization object for *Loop*, but we will not focus on these values and *Prolog* in this chapter.

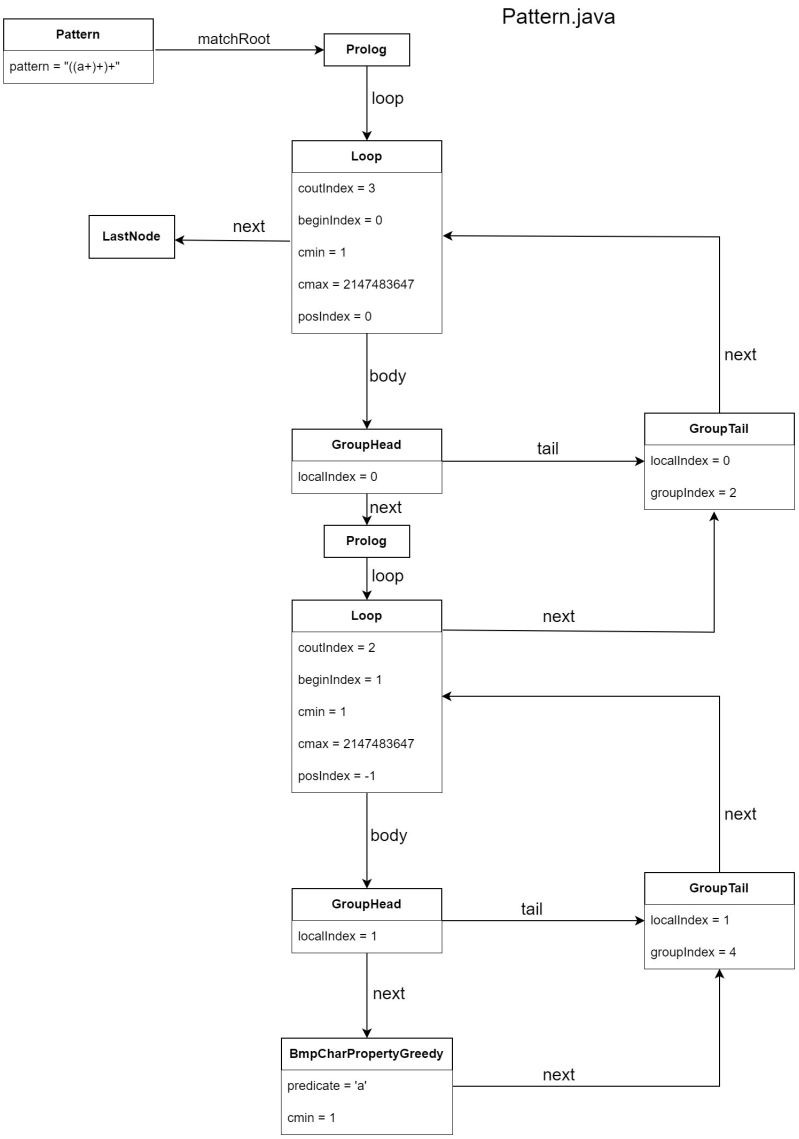

Figure 3.3 Constructed object diagram for vulnerable pattern $((a+)+)+$

For pattern $((a+)+)+$ an additional set of these three objects was created - *Loop*, *GroupHead* and *GroupTail* - these represent our additional capturing group of patterns (see diagram explanations in figure 3.2).

In this subchapter we created a node tree for patterns with Kleene plus, the same trees are created for patterns with Kleene star, for example $((a*)*)*$, except for *cmin* value in *Loop* object equal to zero.

3.2.2 Pattern $(a+)\{1,100\}$

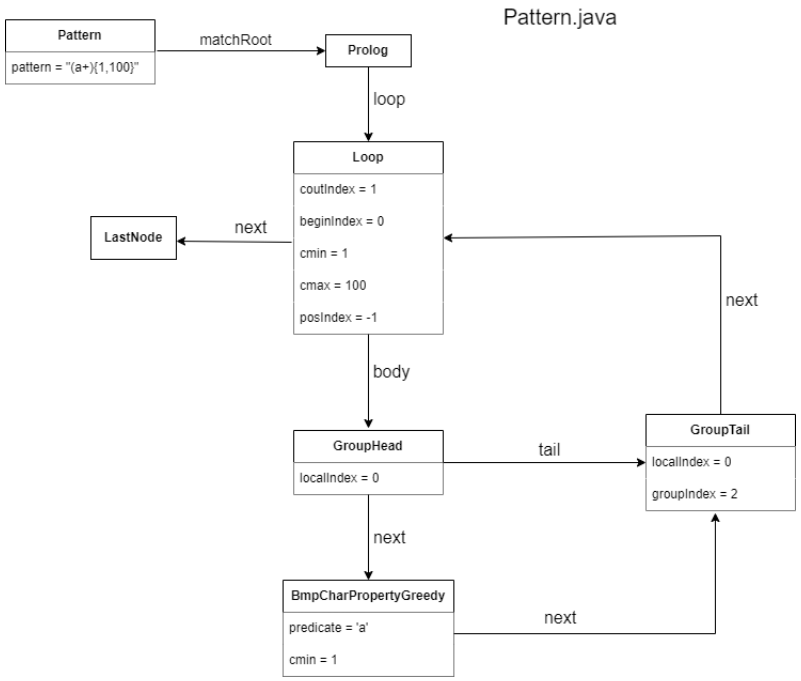

Figure 3.4 Constructed sub classes diagram for vulnerable pattern $(a+)\{1,1000\}$

For this pattern a similar data structure was created as in figure 3.2, except for some small differences, $posIndex$ is set to -1 and $cmax$ is set to 100.

3.2.3 Patterns $(a|a)\{1,100\}$ and $(a|a)+$

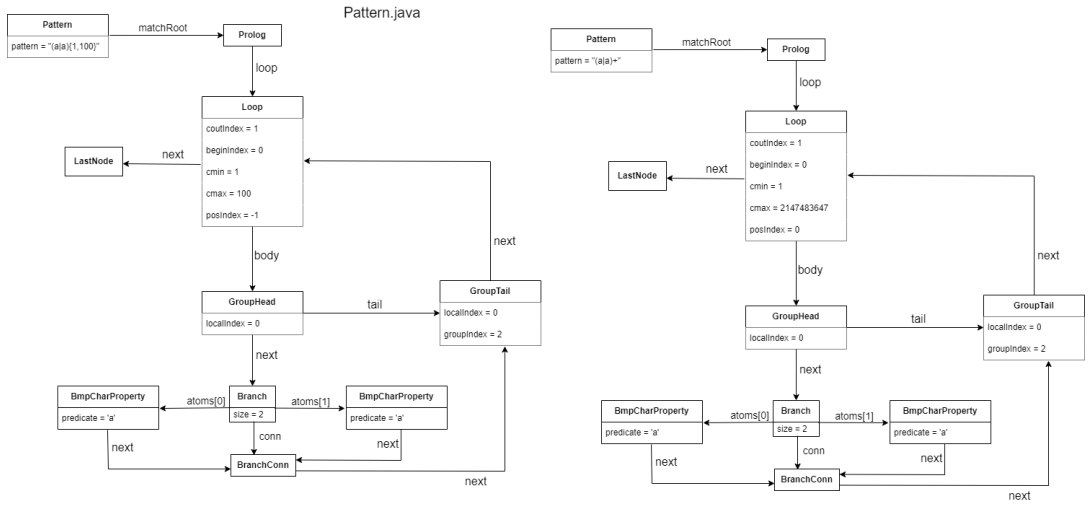

Figure 3.5 Constructed object diagram for patterns $(a|a)\{1,1000\}$ (left) and $(a|a)+$ (right)

In these diagrams two new objects can be seen, *Branch* and *BranchConn*. These objects represent the set of alternatives due to the symbol "|" within their pattern. These objects hold two pointers to the *BmpCharProperty* nodes (our two alternatives - '*a*'s).

3.2.4 Pattern $(a+) + \backslash 1$

For our vulnerable backreference pattern, there is no need to make a diagram, as the tree that is built is nearly identical to the one seen in figure 3.2, but with two small points of difference. For this pattern when the tree is created, the variable *posIndex* is set to -1 indicating that backtracking is enabled, and in place of the *lastNode* object that can be seen, an object called *backRef* is placed and that object points to *lastNode*

3.3 Short pseudo-codes of match method

In this subchapter, we will dive into the inner workings of the 'match' method in the OpenJDK regex engine, providing a concise yet insightful overview of its pseudocode implementation.

Understanding the pseudocode of the *match* method is essential for gaining a deeper comprehension of how the OpenJDK regex engine handles pattern matching behind the scenes. By examining the pseudo-code, we can uncover the core algorithms and steps involved in the matching process.

We will explore the pseudocode of the 'match' method within different child node classes. We will do so in a clear and concise manner by breaking down the algorithm into its essential steps and explaining the purpose and significance of each. By doing so, we can truly see where these vulnerabilities and issues originate. So, let us dive in and explore the fascinating pseudocode of the *match* method.

Algorithm 1 Match methods' pseudo-code in the *Loop* class

Require: $i \wedge seq$
Ensure: $i \leq seq.length$
1: **if** $posIndex \neq -1 \wedge isFailedBefore(i)$ **then**
2: **return** $next.match(i, seq)$
3: **end if**
4: **if** $body.match(i, seq)$ **then**
5: **return true**
6: **end if**

The above pseudocode is the algorithm displayed within the match method, which is located in our *Loop* object that can be seen in all our diagrams. As you can see, the method takes in two parameters, "i" – which is showing us the algorithm's current index within the given input string and 'seq', which represents the input string itself. On line 1 we simplify the origin logic and replace it with method $isFailedBefore(i)$, emphasizing here if a failure match was performed before on index "i".

It is important to note that the pseudocode above was simplified a lot to convey a specific point. The real match method is more complex and includes the *Matcher* object and its parameters.

Algorithm 2 Match method in *BmpCharPropertyGreedy* class

Require: $i \wedge seq \wedge (N = 0)$
Ensure: $i \leq seq.length$
1: **while** $seq[i] =' a'$ **do**
2: $i \leftarrow i + 1$
3: $N \leftarrow N + 1$
4: **end while**
5: **while** $N \geq cmin$ **do**
6: **if** $next.match(i, seq)$ **then**
7: **return true**
8: **end if**
9: $i \leftarrow i - 1$
10: $N \leftarrow N - 1$
11: **end while**
12: **return false**

This pseudocode represents the algorithm shown within the object's *BmpCharPropertyGreedy* match method. When the regex engine creates the node tree, best seen in the figure 3.1, for patterns with Kleene plus this object is used to compare the input characters with our pattern "a" (line 1). After the comparison, we enter a backtracking *while* loop (line 5) and enter the next node (line 6) with every iteration. In the worst-case scenario, the algorithm might try checking n^2 times, with n representing the input length.

3.4 Analysing cause of exponential executions

In first glance at the diagrams, we can see the loops between objects *Loop*, *GroupTail*, and the object that is used to predicate the symbol or string (for example in figure 3.4 is the *BmpCharPropertyGreedy* node). And from here we can immediately notice the danger not only from time execution but also from stack overflow when we use recursive calls from these objects. For example, when we look at figure 3.4 and the behavior of the algorithm in pseudocodes 1 on line 4 and pseudocode 2 on line 6, we can see that we have a complex recursion that calls each other in a while loop (2 line 5). Based on this, a stack overflow can occur if we have a sufficiently large input.

In the pseudocode of the match method in the *Loop* object if pseudocode 1, we can see that if *posIndex* is set to -1, the algorithm first goes to the *body* node and if it fails, it continues to the *next* node. This can explain the behavior in figure 2.1, where we had an extremely large growing run-time with each "()+" added to the pattern. When we add an additional combination of a capture group and Kleene plus - "()+", we are creating more *Loop* objects (see in figure 3.2 and figure 3.3), which are wired. Each of them first goes to the *body* node and then to the *next* node, which is another *Loop* object, and continues to another *body* node.

Furthermore, in the *Loop* pseudocode (pseudocode 1) on line 1 we can find the optimization that was added in version 9. Optimization allows us to disable backtracking to avoid excessive run-time with patterns like $(a+)+$. A check was added within the *Loop* objects *match* method that checks if we failed to match a character to our pattern before the current index, "i", of our input string and if the backtracking has been disabled (*posIndex* not equal to -1). If this is true, then we are not continuing to the *body* node, but straight to the *next* node (see the pseudocode 1 line 2), thus significantly shortening the execution time. This can explain why patterns such as $(a+)+$ and $(a+)\{1, 100\}$, which have very similar behaviors and accept almost the same input, have such different run-time. When we look at the figure 3.2 and figure 3.4, we can see that for pattern $(a+)+$ the backtracking is disabled (*posIndex* equal to 0), but for pattern $(a+)1, 100$ it is enabled (*posIndex* equal to -1). The same situation can be found in figure 3.3 and figure 3.5.

In the pseudocode of the *match* method in *BmpCharPropertyGreedy*, pseudocode 2, on line 5 we can find a *whileloop* with the condition $n > cmin$ where the "n" value is set to zero (according to the algorithm' requirements). This can explain why in graphs with patterns including Kleene stars (figure 2.2) the execution time is greater than in patterns including Kleene plus (figure 2.1). In patterns of Kleene stars, *cmin* value is set to zero, which means that in *whileloop* on line 5 we enter an addition one time to the *next* node.

3.5 Conclusion

In conclusion, the analysis of the causes behind the exponential run-time behavior in vulnerable patterns provides information on the challenges and potential solutions to improve the efficiency of the regex engine.

Through our examination, we found several key factors contributing to the exponential run-time behavior, including the complexity of regex patterns, inefficient quantifiers, or capturing groups.

In our next chapter, we will look into different theoretical solutions to the oversights within OpenJDK that we looked into within this chapter.

Chapter 4

Possible theoretical solutions

This chapter explores possible theoretical solutions in relation to the exponential run-time in the regex engine within OpenJDK. By proposing alternative approaches and optimizations, we aim to mitigate the impact of this behavior and improve the overall efficiency of regex processing. We will take into account all our experimental findings, documented in Chapters 2 and 3, to assess what possible solutions can be explored to eliminate or reduce the risk of a ReDoS attack to systems using the OpenJDK regex engine.

4.1 Deconstruction of the problem behind OpenJDK's regex engine

At this point, we can confidently point out the main issues with the OpenJDK implementation and the root causes of the vulnerable criteria that we defined in section 2.2. We can see that as a whole, the experiential runtime allows the engine to consume more and more resources, as can be seen from our subchapter 3.4, which analyzed the exponential runtime. The analysis also showed that this consumption of resources was caused by the backtracking itself within these patterns. This is due to the tree these patterns build; which is a result of the NFA implementation. Due to the nondeterminization of the algorithm, large trees are built, as can be seen in the figures 4.2, 3.3, 3.4, and 3.5, which results in much more backtracking needed.

With this in mind, we can go step by step to see what solutions would best mitigate these problems, starting from the quickest and simplest theoretical solutions to the more lengthy and total ones.

4.2 Timing-out

When looking at the exponential run-time issue as a whole, a very simple and obvious solution stands out; adding a timeout to our regex engine. Though this addition does not fix the inner-laying problem of the engine, it will cause it to no longer be vulnerable to ReDoS attacks.

This kind of solution could be added as an optional feature. For example, the timeout will be disabled by default, but can be enabled when called by giving the call an additional variable within its parameters; this variable will be the timeout length and by default will be equal to $NULL$. The developer would set the time-out in milliseconds, putting a variable holding a numerical value in the designated parameter within the call for the matching algorithm. From there, the matching sequence within the regex engine would run for a maximum amount of

milliseconds designated by that given value before it returns an exception. This allows us to limit the run-time and makes sure we do not get stuck in our matching process.

This implementation of timing-out could be done by adding an additional scheduler thread that will be created when the matching algorithm starts the execution process, for example in the method *matches*() in the *Matcher* class; mentioned in section 3.1. This thread will check at specified times using an atomic value that would be located within the class *Matcher* if the execution is completed or not. If the execution is not done in the specified time, the thread will set another atomic value to stop the matching algorithm and terminate itself. This atomic value will be checked in all *match*() methods in the *Pattern* class and throw the exception if it was set to stop.

A similar solution was described in "Preventing Unbounded Regular Expression Operations in Java" [21], where it describes a theoretical solution of setting a timeout in the *charAt*() method. The proposed solution was due to an investigation that determined that the method *charAt*() was the most accessed method within OpenJDK's regex matching algorithm. This would allow the timeout to accrue at the lowest level of the algorithm and only be called if we have reached a potentially dangerous scenario. The document shows the implementation of this solution. OpenJDK investigated this idea, though they decided to not implement it with the response being posted on there Bug System [22] stating that *"The problem is that this approach relies on the engine internals. It will break if tomorrow the engine, for example, will operate on some array where the input was copied, and thus will stop invoking the override charAt."*.

4.3 Expand disabling backtracking

Since OpenJDK version 9, there are possibility to enable and disable backtracking as can be seen in figure 1. Disabling backtracking drastically reduces runtime for these types of vulnerable patterns, as can be seen in figure 2.5, and eliminates the possibility of catastrophic backtracking. Disabling backtracking currently works for a small group of patterns like (a—a)+ or (a+)+, but for similar patterns that have manual repetitions instead of the kleene plus or star backtracking is still enabled, as can be seen within figure 3.4 with the variable *posIndex* being equal to -1. The disabling of backtracking could be expanded for this group of patterns.

Additionally, this kind of solution can be expanded to other groups of vulnerable patterns such as combinations of capturing groups and quantifiers, such as pattern $((a+)+)+$, $(((a+)+)+)+$ or patterns with combinations of capturing group and backreferences $(a+) * \backslash 1$.

When investigating the tree that each of our patterns creates in section 3.2 it is not fully clear what criteria are currently in place to determine whether backtracking should be disabled. From what we could analyze from our findings, it seems that when a kleene plus or star, by itself and not connected to any other sub-string, is placed after a single capturing group, then the pattern fits the criteria for backtracking to be disabled. With the addition of more capturing groups placed around the original capturing group, or a manual repetition being in the place of the kleene plus or star, the pattern no longer falls under the needed criteria for backtracking to be disabled, and therefore can be subjected to catastrophic backtracking and experiential run-time problems. If we expand the criteria to include patterns that have manual repetition in the place of kleene plus or star, and patterns that can easily simplify to our no longer vulnerable patterns, such as $(a+)+$, we can greatly reduce the number of potentially dangerous patterns that exist for OpenJDK's regex engine. This kind of solution will not solve all incoming problems related to exponential run-time, but it will make the OpenJDK regex engine more secure.

4.4 Safe regex implementation

Our most complex solution would also be our most thorough one; this would be a complete reworking of the regex engine. As we mentioned before in subsection 1.27, the current OpenJDK

regex engine's algorithm builds NFA's for each pattern, rather than the safer DFA based solution. This is mainly because NFA's can cover regex features that a DFA implementation cannot, but its important to note that most regex patterns can be covered by a DFA based regex engine implementation. For this reason, creating a safe regex implementation should not replace the current engine but could be added in the same regex package.

This safe regex engine would be less powerful than the current one in the sense that it would not cover all current features the OpenJDK engine providers, as mentioned in subchapter 1.27, but it would be safe and guarantee a linear run-time of the matching process. The safe regex can be used for validation of input texts from user in web applications to avoid any malicious attempts of inflicting a ReDoS attack. In some cases the two engines can even be combined, as suggested in [23], in a solution where all patterns would first attempt to be built as DFA's, and only the patterns that do not succeed in this transformation will be sent to our original regex engine to be built as NFA's.

This would be a substantial improvement to the current regex implementation. This can be best seen when looking at the automaton built for our now no longer vulnerable pattern $(a+)+$, as shown in [15].

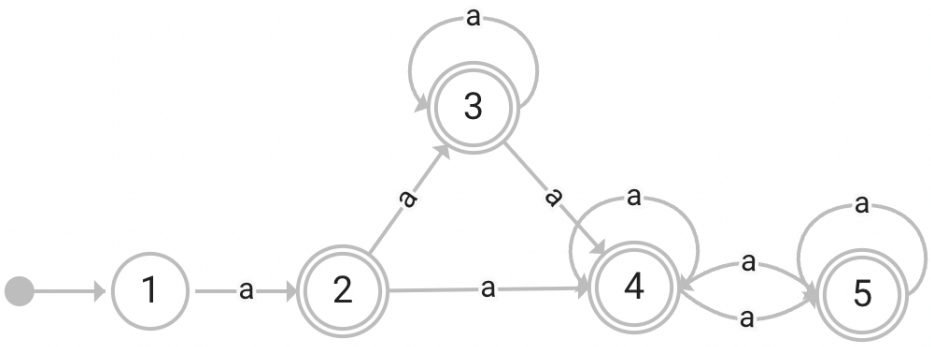

Figure 4.1 NFA for pattern $(a+)+$

This figure is the automata representation of the tree that is built by the regex engine in OpenJDK, as shown in figure 2.1. As you can see in the NFA figure 4.1 we have a lot of repetitive transitions that can cause a lot of nondeterministic movement, when needing to backtrack this movement, it is no wonder the memory consumption and run-time are so large.

When turning this diagram into a DFA model, we will see the extent of the simplification that can be done to this automaton. To turn this NFA into a DFA, we have used the method of determinization and DFA minimization to our automaton taking into account the capturing groups.

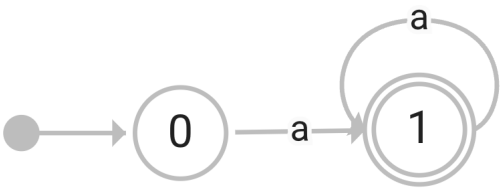

Figure 4.2 DFA for pattern $(a+)+$

Here we can see in our above figure 4.2 that when converting to the automaton to a DFA in this instance we can keep all the used regex features and create a non-repetitive and safe automaton for our regex engine to use. The result in this figure is not only the DFA implementation of the pattern $(a+)+$, but due to the simplification of the automata it is also a representation of the pattern's $a+$ tree, as can be seen in figure 3.1.

A DFA regex engine implementation is not a new concept, in fact, this kind of implementation already exists, for example, Google's RE2 engine [24], which is a DFA-based implementation that guarantees O (n) performance. Even the OpenJDK feature draft [23] mentions the RE2 engine as a proposed addition.

Though this solution will not protect OpenJDK's regex engine from all possible patterns when the two engines are combined, it will help reduce the possibilities by the most significant amount.

Conclusion

In conclusion, throughout this thesis we have explored and dissected the topic of vulnerable regular expressions within OpenJDK. Through our analysis we have shed light on the inner causes of these patterns bad characteristics and have seen the run-time results of their use. With the help of research papers, online findings, testing, experiments, and debugging, we were able to present several key findings on the topic and achieve the goals we have set in place.

We can now be sure that we uncovered and shown the root causes of the vulnerabilities regex can cause that we have aimed to examine; experiential run-time, and catastrophic backtracking. Using the information we have obtained, we have been able to observe that these security risks disappear for several previously vulnerable patterns when optimization was added in OpenJDK 9, as discussed in sub-chapter 1.4. The optimization added was the ability for the algorithm to disable backtracking with-in a specific group of patterns. This optimization helps show the effects backtracking can inflict on complex, or risky regex patterns, as can be seen in the figure 2.5. This coupled by the evaluations shown in chapter 3 allows us to understand the amount of backtracking that can occur when a repetitive pattern that includes capturing groups is fed a nonmatching input. Due to the fact that with each additional capturing group more and more class objects are added to the patterns tree, we can understand the reason behind the run-time growing exponentially with every addition of a capturing group, with more backtracking needed in response to this growth. Using this and previous work done by different research papers we have mentioned, we can confidently deduce that the root cause of the catastrophic backtracking is the behavior the engine exhibits during the build of the patterns tree, with it taking the straight forward approach and building a large repetitive non-deterministic tree. This leads us to the exponential run-time which is the result of the large amount of backtracking that needs to be done within the tree.

This deduction was in large part the result of our in-depth investigation of the OpenJDK implementation of the regex engine that can be seen in the chapter 3. This in-depth dive showed us the classes that exist within the engines implementation and their use. We were able to understand that tree the engine creates based on the pattern it is given and the way it processes said pattern. This insight into the algorithm helps us to understand how it separates and processes the pattern, which in turn helps us to understand how it attempts to match the given inputs. The match process was deconstructed and shown in our presented pseudocodes 1 2 where we could see that each child node object has its own match process in place but with a similar methodology.

Within this thesis we were also able to find a number of vulnerable patterns, as seen in chapter 2, we were able to prove their vulnerable status by providing experimental evaluations and compare their run time, not only to one another but also to nonvulnerable patterns. Our experiments consisted of multipulse inputs with a wide range of lengths and can be best seen in their corresponding graphs in section 2.4. When conducting our experiments we heavily focused

on ensuring that there was no outside influence on our results and that we only measured the effects of the regex engine process.

In the end using everything collected we were able to theorize about possible solutions that can be implemented to help mitigate the possibility of ReDoS attacks to applications using OpenJDK's regex engine. These solutions were based on our own analysis as well as proposed solutions from other research papers. When looking at the solutions, we included both the benefits as well as the possible downfalls of them and also added OpenJDK's response to them.

Given our analysis, it can be seen that vulnerable regular expressions still exist within OpenJDk, and have the potential to cause a lot of harm with their use. Hopefully the research done can be used in the future to better implement or improve the way the regex OpenJDK engine works and help reduce its risk factor. Although solutions have been proposed, for now it is the responsibility of each developer to make sure they are not opening themselves to a potential attack. This thesis has shown the importance of developer awareness and education on regex vulnerabilities. By promoting best practices, providing training, and increasing knowledge dissemination, software developers can enhance their understanding of regex pitfalls and proactively address potential security issues during the software development life cycle.

Bibliography

1. WÜSTHOLZ, Valentin; OLIVO, Oswaldo; HEULE, Marijn J. H.; DILLIG, Isil. Static Detection of DoS Vulnerabilities in Programs that Use Regular Expressions. In: LEGAY, Axel; MARGARIA, Tiziana (eds.). *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 3–20. ISBN 978-3-662-54580-5.
2. RAMEL, David. *Malformed stack overflow post chokes regex, crashes site*. 2016. Available also from: `https://adtmag.com/Blogs/Dev-Watch/2016/07/stack-overflow-crash.aspx`.
3. ORACLE CORPORATION. *OpenJDK 20* [comp. software]. 2023. [visited on 2023-06-14]. Available from: `https://openjdk.org/projects/jdk/20/`.
4. ORACLE CORPORATION. *OpenJDK 9* [comp. software]. 2017. [visited on 2023-06-14]. Available from: `https://openjdk.org/projects/jdk9/`.
5. ŠESTÁKOVÁ, Eliška. Finite automata. In: *Automata and grammars: A collection of exercises and solutions*. České vysoké učení technické v Praze, 2018.
6. KLEENE, S. C. Representation of events in nerve nets and finite automata. *Automata Studies. (AM-34)*. 1956, pp. 3–42. Available from DOI: `10.1515/9781400882618-002`.
7. Oracle Corporation, 2023. Available also from: `https://openjdk.org/`.
8. STUBBLEBINE, Tony. *Regular Expression Pocket Reference: Regular Expressions for Perl, Ruby, PHP, Python, C, Java and. NET.* " O'Reilly Media, Inc.", 2007.
9. 2023. Available also from: `https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/util/regex/Pattern.html`.
10. [N.d.]. Available also from: `https://docs.oracle.com/javase/tutorial/essential/regex/groups.html`.
11. BERGLUND, Martin; DREWES, Frank; VAN DER MERWE, Brink. Analyzing catastrophic backtracking behavior in practical regular expression matching. *arXiv preprint arXiv:1405.5599*. 2014.
12. HRON, Martin. *Backreferences in practical regular expressions*. 2020. Available also from: `https://dspace.cvut.cz/bitstream/handle/10467/87858/F8-DP-2020-Hron-Martin-thesis.pdf`.
13. Oracle Corporation, 2014. Available also from: `https://openjdk.org/projects/jdk8/`.
14. Wikimedia Foundation, 2022. Available also from: `https://en.wikipedia.org/wiki/ReDoS`.

15. WEIDMAN, Adar. *Regular expression denial of service - ReDoS*. [N.d.]. Available also from: https://owasp.org/www-community/attacks/Regular_expression_Denial_of_Service_-_ReDoS.

16. 2018. Available also from: https://stackoverflow.com/questions/53048859/is-java-redos-vulnerable.

17. TUROŇOVÁ, Lenka; HOLIK, Lukáš; HOMOLIAK, Ivan; LENGÁL, Ondřej; VEANES, Margus; VOJNAR, Tomáš. Counting in Regexes Considered Harmful: Exposing ReDoS Vulnerability of Nonbacktracking Matchers. In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, 2022, pp. 4165–4182. ISBN 978-1-939133-31-1. Available also from: https://www.usenix.org/conference/usenixsecurity22/presentation/turonova.

18. BERGLUND, Martin; DREWES, Frank; MERWE, Brink van der. Analyzing catastrophic backtracking behavior in practical regular expression matching. *Electronic Proceedings in Theoretical Computer Science*. 2014, vol. 151, pp. 109–123. Available from DOI: 10.4204/eptcs.151.7.

19. Oracle Corporation, 2023. Available also from: https://docs.oracle.com/javase/8/docs/api/java/util/regex/Matcher.html.

20. JetBrains, 2023. Available also from: https://www.jetbrains.com/idea/.

21. REASON. *Preventing Unbounded Regular Expression Operations in Java*. 2017. Available also from: https://www.exratione.com/2017/06/preventing-unbounded-regular-expression-operations-in-java/.

22. WANG, Andrew. *Simple redos prevention*. 2021. Available also from: https://bugs.openjdk.org/browse/JDK-8268854.

23. BUCHHOLZ, Martin. *Predictable regex performance*. 2022. Available also from: https://openjdk.org/jeps/8260688.

24. GOOGLE. *Google/RE2*. 2023. Available also from: https://github.com/google/re2.

Contents of enclosed CD

- `/`
  - `src`
    - `test` ..... test scripts and Java binaries
    - `result` ..... results of tests organized in excel tables
    - `thesis` ..... the directory of LaTeX source codes of the thesis
  - `text` ..... the thesis text directory
    - `thesis.pdf` ..... the thesis text in PDF format