Availability Vulnerabilities
Report for the *Program Analysis for Vulnerability Detection* Seminar

Mitul Bipin
Saarland University
mibi00001@stud.uni-saarland.de

***Abstract*—An algorithmic complexity vulnerability significantly exhausts the resource of a server by pushing the application to exhibit its worst-case behavior. This vulnerability can often lead to performance slowdowns or even a Denial-of-Service attack. We can observe algorithmic complexity attacks in web/mobile implementations that have a weak code logic or in web/mobile applications that do not rely upon PaaS platforms. Such attacks can be majorly caused due to vulnerable regular expression patterns that can exhaust the system resources by processing a malicious string input.**

In this report, we focus on two approaches that aim at detecting and exploiting algorithmic complexity vulnerabilities in a wide variety of applications. The approaches are based on static and dynamic analysis. While the approaches meet their expectations, executing them separately can be time-consuming. Therefore, the report also proposes a hypothesis to combine the two approaches and obtain their findings together. The combination of two approaches can significantly reduce the number of hours spent on confirming the presence of an algorithmic complexity vulnerability.

***Index Terms*—Algorithmic complexity attacks, ReDoS attacks, extended regular expressions.**

I. Introduction

An algorithmic complexity vulnerability is caused by inputs that trigger a repetitive processing loop within the system. Such a vulnerability can exhaust the CPU usage of the target system. An adversary can launch a DoS attack by passing malicious inputs that trigger the worst-case behavior. Manually detecting the vulnerability in an application can be difficult because the time complexity significantly changes with minor variations in the algorithm implementation.

A Regular Expression Denial-of-Service (ReDoS) is a type of algorithmic complexity vulnerability targeting the CPU usage of a server that makes use of regular expressions for input validation. To verify a user input, developers may unknowingly use regular expressions that exhibit super-linear/ exponential worst-case matching time. As the leading mobile/web applications rely on regular expressions to validate user inputs, the ReDoS vulnerability can have a significant impact on them. In this report, we are going to discuss the generation of ReDoS vulnerability with extended regular expressions only.

Two well-known methods of finding bugs - static analysis and dynamic analysis are popular among developers and testers. Static analysis often uses parse structures [1] to detect vulnerable patterns. These structures do not support all extended features. Moreover, detecting vulnerable patterns in regulation expressions with extended features is not theoretically well studied. Fuzzing is used to dynamically detect ReDoS. However, fuzzers cannot find the inputs/regular expressions that exhibit super-linear matching time in extended regular expressions. Moreover, dynamic generation of inputs sometimes spend a lot of time on generating inputs that may not trigger the vulnerability. REVEALER, the first approach, combines static and dynamic analysis to find ReDoS attacks. REVEALER detects the algorithmic complexity vulnerability in regular expression patterns that are generated by Java-based regular expression engines. For a given vulnerable regular expression pattern, REVEALER can generate attack strings that exhibits the super-linear matching time.

Algorithmic complexities can occur, not just on Java-based regular expression engines, but also on a broader range of the spectrum that includes compression utilities, hash tables, and modern applications that rely on regular expressions for input validation. SlowFuzz, the second approach, is a domain-independent framework that discovers algorithmic complexity vulnerabilities on the aforementioned platforms. The framework is highly driven by the evolutionary searching technique that generates inputs to trigger the worst-case behavior.

The two approaches aim to detect and exploit the same vulnerability, but their outcomes are different. REVEALER discovers attack strings in extended regular expressions, while SlowFuzz generates traditional regular expressions exhibiting super-linear matching time. We can combine the two approaches to generate attack strings and vulnerable regular expression patterns at the same time. Implementing such an approach can save the efforts of executing the two tools separately.

The structure of the report is as follows. The report briefly explains how REVEALER detects and exploits ReDoS vulnerability in Section II. Section III describes the architecture of SlowFuzz with an example of Quicksort. A comparative analysis highlights the key differences between the two approaches is outlined in Section IV. Section IV also includes an approach that combines both REVEALER and SlowFuzz. Finally, the report is concluded in Section V.

II. Paper I (REVEALER: A tool to detect ReDoS vulnerabilities [1])

The paper presents a tool, REVEALER, that detects and exploits ReDoS vulnerability, that is a type of an algorithmic complexity attack. REVEALER combines both static and dynamic analysis to generate an attack string that exhibits super-linear matching time.

A. Introduction

REVEALER is based on a Java 8 regular expression engine and adapts to a hybrid approach by including both static and dynamic analysis. The former is responsible to discover vulnerable sub-expressions while the latter produces attack strings and validates potentially vulnerable patterns. REVEALER is evaluated against data-sets containing over 29,088 regular expressions, and it outperforms ReScue [4], RXXR2 [5], Rexploiter [6], the three state-of-the-art tools by detecting over 213 unknown ReDoS vulnerabilities [1], and all 237 known ones that have been detected by the other tools.

B. Modelling ReDoS Vulnerabilities

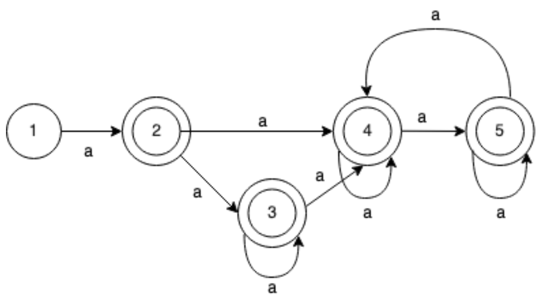

Figure 1: A transition diagram representing non-deterministic finite futomata (NFA).

The ReDoS vulnerability contains a set of potentially vulnerable matching path(s) that are responsible for exhibiting super-linear matching time. Figure 1 represents a transition diagram that has multiple ending states. As the modern regular expression engines support extended features, a vulnerable regular expression pattern can catastrophically exhaust a system's resource. In an ideal scenario, the engine tries to find a match as long as it can go back to the previous positions exhibiting the worst-case time complexity $O(n^2)$. For example, a regular expression (a+)+, whose transition diagram is shown in Figure 1, contains a vulnerable sub-expression (a+). For the input *aaaaX*, there are 16 possible paths. However, for *aaaaaaaaaaaaaaaaX*, there are over 65,536 possibilities, and the number is double for every additional *'a'*. The root cause for the issue is in the regular expression engine feature known as *Backtracking*. The feature allows the regular expression engine to go back to the previous position in search of a different path if the input fails to match.

A vulnerable regular expression pattern consists of a loop and/or branch states. Such patterns often lead to super-linear matching time.

C. REVEALER

REVEALER accepts a regular expression pattern as e-NFA from a Java-based regular expression engine. The regular expression engine generates a potentially vulnerable regular expressions that are predicted to exhibit super-linear matching time. First, the vulnerable patterns are identified in the e-NFA in the static analysis phase. The vulnerable patterns are identified by passing the regular expression through the *E-TREE* and *Vulnerable Structure Detection*. Second, the attack string(s) are generated in the dynamic analysis phase using the identified vulnerable patterns. To reduce the constraint of obtaining the attack string, the attack string length and minimum matching step count are predetermined. Finally, based on the matching string s *(attack core)*, the attack prefix (s_0) and attack suffix $(s_|)$ are generated to obtain the final attack string.

The static analysis discovers a set of vulnerable patterns. It achieves the objective by converting the regular expression pattern in the form of an E-TREE [1]. An E-TREE represents each state in a hierarchical manner that reduces the complexity of searching a specific state within the regular expression. In a traditional regular expression representation, all the states are equally treated making it hard to trace the traversal order. Finally, the vulnerable patterns are written by the vulnerability detector that traverses across the E-TREE. The vulnerable pattern *P* is in the format *P* = $< v_0, w_0 >, < v_1, w_1 >, ..., < v_n >< w_n >$, where *v* is a loop state and *w* is a loop/branch state.

The dynamic analysis verifies whether a vulnerable pattern $p \in P$ contains a common match string *s*. This is achieved by utilizing two algorithms that are included in the dynamic analysis phase. *1) Single Match Algorithm*: Generates an attack core *s* for a single matching path. *2) Common Match Algorithm* Generates an attack core *s* for multiple matching paths.

The attack prefix s_0 and attack suffix s_1 based on the attack core *s* obtained from the dynamic analysis phase. It is important to note that a valid suffix must fail the regular expression match throughout all the possible matching paths. Finally, an attack string is formed by combining attack prefix s_0, attack core s, and the attack suffix s_1 in the format s_0, s^k, s_1 where k = $\frac{l_m - length(s_0) - length(s_1)}{length(s)}$ [1].

D. Evaluation

REVEALER is evaluated along with three tools that are well known for detecting ReDoS vulnerabilities. 1) ReScue [4], 2) RXXR2 [5], 3) Rexploiter [6]. Second, a dataset consisting of 29,088 regular expressions from [4] has been applied on REVEALER to check its effectiveness in detecting ReDoS vulnerabilities.

TABLE I
EVALUATION RESULTS OF REVEALER

Tool	No. of Vul	No. of FP	Error Rate(%)	Avg. Time(s)
REVEALER	450	0	0.00	0.0076
ReScue	187	0	0.00	18.2259
RXXR2	112	103	47.91	0.0042
Rexploiter	63	1959	96.88	0.4472

The overall results are shown in Table I. Over 812 ReDoS vulnerabilities have been detected by the four tools combined. REVEALER has significantly outperformed the three tools by detecting 450 ReDoS vulnerabilities with an average time of 0.0076 seconds. Both ReScue and REVEALER did not report any false positives. However, ReScue takes much more average time than REVEALER, and it detects significantly fewer vulnerabilities than REVEALER. Both RXXR2 and Rexploiter clocked under 0.5 seconds of average time. REVEALER shows that it can detect ReDoS vulnerabilities better than its counterparts with no error rates, false positives, and a negligible average time. Moreover, REVEALER can find vulnerable regular expression patterns on extended regular expressions that support extended features.

III. PAPER II (SLOWFUZZ: DETECTION OF ALGORITHMIC COMPLEXITY VULNERABILITIES) [2]

Paper I majorly deals with detecting regular expressions leading to ReDoS vulnerabilities based on Java regular expression engines. It is not feasible to develop multiple approaches to find a similar vulnerability on mobile/web platforms.

A. Introduction

SlowFuzz is a domain-independent framework that is fully automated to find Algorithmic Complexity Vulnerabilities. The tool can generate crafted inputs until the worst-case behavior of the application is triggered without requiring any domain-specific rules.

B. Applying SlowFuzz on Quicksort Algorithm

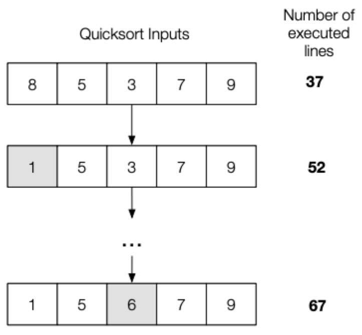

Figure 2: SlowFuzz's input generation mechanism

Let us consider Quicksort, a well-known sorting algorithm widely known among developers. Quicksort shows an average time complexity of $O(nlogn)$ and a worst-case complexity of $O(n^2)$. It is important to understand the pivot selection mechanism to trigger the worst-case behavior by giving relevant input. However, SlowFuzz would generate inputs without knowing the pivot selection mechanism.

To further illustrate this point, let us assume the initial corpus of SlowFuzz consists of a single array A = [8,5,3,7,9] [2]. At each step, SlowFuzz randomly selects a set of input from the input corpus, passes it to the Quicksort algorithm, and mutates the input(s) till the algorithm shows its worst-case behavior. While the tool passes several inputs, it records the number of executed statements. As shown in Figure 2, the inputs [8,5,3,7,9] results in 37 lines of execution. The inputs are further mutated to [1,5,3,7,9] that executes 52 lines of code, which is marginally higher than the first input. The grey box represents the mutated input during execution. Eventually, SlowFuzz will generate a fully sorted array [1,5,6,7,9] that will demonstrate the worst-case behavior in the Quicksort implementation with 67 lines of code.

C. Architecture

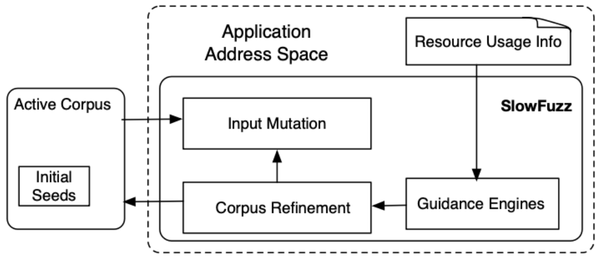

Figure 3: SlowFuzz Architecture

SlowFuzz is inherited from LibFuzzer [7], a popular fuzzing tool, by modifying approximately 550 lines of C++ code. Similar to LibFuzzer, SlowFuzz is executed within the same address space as the test application. The test application is instrumented to allow SlowFuzz to access different resource usage metrics. As shown in Figure 3, a set of inputs are passed into the test application through the active input corpus. SlowFuzz constantly refines the inputs during execution based on the *Fitness Function* [2]. In every iteration, an input is selected from the corpus, it is mutated, then passed into the test application for execution. After each execution, SlowFuzz ranks the inputs based on the score assigned by the fitness function. The highest-ranked (based on the highest score) inputs are subject to further mutations and passed into the test application for further execution. SlowFuzz continuously generates inputs until the test application demonstrates its worst-case behavior.

A fitness function is equivalent to the guidance engine in Figure 3. It is implemented to rank every input(s) after execution. Based on the ranking, a score is generated that is passed to SlowFuzz to determine whether the input unit should be mutated further. The function is also responsible to record the system resource to gauge the impact of the input. The hypothesis is that the test application becomes slower as the inputs are further mutated.

The mutation operations are inherited from LibFuzzer [7] with minor internal modifications. SlowFuzz introduces several strategies to mutate a variety of inputs that are passed to domain-independent test applications. A mutation strategy decides the mutation operation and the byte offset in an input to modify. 1) Random Mutations: As the name suggests, one of the mutation operations is selected at random and applied to input until it does not hinder the parameters that are set before the testing session. 2) Mutation Priority: A mutation is selected with s probability based on its previous success of producing slow units. Initially, a mutation operation is picked at random with $(1-e)$ probability. 3) Offset Priority: The byte offset chosen is based on the prior success in generating a large number of executed instructions. 4) Hybrid: The final strategy combines both Mutation and Offset Priority. By obtaining a specific mutation strategy, SlowFuzz can achieve modified bit/bytes in the input in a random order.

D. Evaluation

The evaluation of SlowFuzz is performed on a 64-bit Debian 8 powered by an Intel(R) Xeon(R) CPU X5550 @ 2.67Hz with 23GB RAM. The tool is compiled with GCC v4.9.2 with a kernel v4.5.0. First, SlowFuzz is applied to popular sorting algorithms whose worst-case time complexity is well-known. Second, the tool is applied to a wide variety of applications that are known to be vulnerable to algorithmic complexity attacks. The PCRE regular expression library, PHP hash table, and a compression utility have shown the worst-case behavior. Table II shows that SlowFuzz successfully induces a slowdown on various domain-independent platforms. SlowFuzz achieves a *5.12x* slowdown in a Quicksort implementation that is explained in Section III.B. The slowdown is caused by the inputs generated by SlowFuzz.

Moreover, SlowFuzz induces 8%-25% slowdown in the PCRE regular expression library without knowing the structure of the regular expression engine. The slowdown is caused due to regular expression patterns that cause a super-linear matching time by passing a malicious input.

TABLE II
EVALUATION RESULTS OF SLOWFUZZ

Tested Application	Outcome
Quicksort	5.12x slowdown
Insertion Sort	41.59x slowdown
PCRE (fixed regular expression)	8%-25% slowdown
PHP hash table	20 collisions
bzip2 decompression	300x slowdown

SlowFuzz generates regular expression patterns that exhibit super-linear and exponential matching time. To achieve this, SlowFuzz passes a set of inputs to the PCRE regular expression library. A special character set is reserved to generate regular expression patterns of various types. SlowFuzz is unaware of the structure of the regular expression engine. The generated regular expression patterns are further passed to Rexploiter to verify whether they exhibit super-linear matching time or not.

Overall, SlowFuzz manages to generate 33,343 regular expressions, out of which 6,201 regular expressions are valid. Among the valid regular expressions, 765 trigger super-linear matching time, and 78 trigger the exponential matching time. Out of the 6,201 regular expressions, SlowFuzz triggers 12.3% and 2.3% of regular expressions that invoked super-linear and exponential matching time respectively. By adopting the evolutionary searching technique, SlowFuzz successfully triggers the worst-case behavior on various domain-independent platforms.

IV. COMPARATIVE ANALYSIS

Both REVEALER [1] and SlowFuzz [2] aim to detect and exploit algorithmic complexity vulnerabilities. ReDoS attacks that are detected by REVEALER belong under the algorithmic complexity vulnerability as the attack exhausts the system by leveraging a weakness in the regular expression engine. The two tools are channeled towards a common direction but have minor differences in implementations, design architecture, target platforms, etc. The prototype of REVEALER is based on Java, while SlowFuzz is inherited from LibFuzzer which is based on C++. The former takes a hybrid approach that involves both static and dynamic analysis, while its counterpart generates the worst-case behavior through dynamic analysis only.

Interestingly, SlowFuzz generates over 33,343 inputs (regular expression patterns) out of which only 6,201 [18.60%] are valid. It is important to note that the 6,201 do not trigger super-linear matching time. Out of the 33,343 regular expressions, SlowFuzz only generates 765 [2.29%] regular expressions and 78[0.23%] regular expressions causing super-linear and exponential matching time respectively. The numbers are drastically less as the tool is spending over 82% of the efforts in generating invalid regular expressions. On contrary, REVEALER has outperformed all the three aforementioned tools by detecting over 213 unknown vulnerabilities with an 89.87% improvement over the combined number of vulnerabilities found by Rexploiter, ReScue, and RXXR2.

As SlowFuzz only generates potentially vulnerable regular expressions, the final results of SlowFuzz completely rely upon Rexploiter [6], a tool that detects super-linear matching time in classical regular expressions. We can use this connection to our benefit in combining REVEALER [1] and SlowFuzz [2].

We know that SlowFuzz utilizes Rexploiter [6] to calculate the super-linear/ exponential matching time of potentially vulnerable regular expression patterns generated by SlowFuzz. Rexploiter only supports classical regular expressions that are represented by a context-free grammar. Such grammars do not support the extended features mentioned in Paper I, thus making SlowFuzz incapable of detecting vulnerable extended regular expressions that may exhibit super-linear/exponential matching time. SlowFuzz can instead utilize REVEALER to confirm vulnerable regular expressions patterns. We can achieve the following results at the same time after integrating SlowFuzz and REVEALER.

1) Obtain regular expression patterns from SlowFuzz that support extended features.
2) Generate attack strings in REVEALER for the subsequent regular expression patterns.

This can be achieved by modifying the seed inputs (dictionary of characters) passed into SlowFuzz. The dictionary consists of characters that form a regular expression pattern. SlowFuzz accepts the inputs and obtains regular expressions through the PCRE regular expressions library. Upon obtaining a set of regular expressions, the valid patterns can be passed to REVEALER.

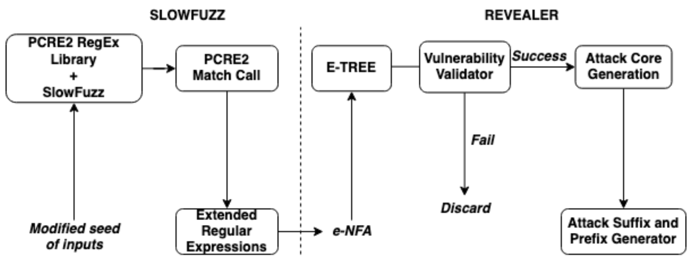

Figure 4: An architecture overview of combining SlowFuzz and REVEALER

As shown in Figure 4, the regular expression patterns obtained in SlowFuzz are represented by e-NFA, as it supports extended features. With the combination of static and dynamic analysis explained in Paper I, REVEALER generates attack strings for all the patterns passed by SlowFuzz. Furthermore, the vulnerable detector removes regular expression patterns that do not generate vulnerable attack strings.

V. Conclusion

The papers presented in the report successfully discover and exploit algorithmic complexity vulnerabilities. While SlowFuzz generates vulnerable regular expression patterns through dynamic analysis, REVEALER produces attack strings that exhibit super-linear matching times in extended regular expressions by combining both static and dynamic analysis.

Detecting the presence of an algorithmic complexity vulnerability on different platforms is hard as modern applications equip complex regular expression patterns that come with extended features. Implementing a separate tool that finds the same vulnerability in each platform can be redundant and it results in wastage of time and efforts. The analysis in Section IV briefly shows an architecture of how SlowFuzz and REVEALER can be combined to detect vulnerable regular expression patterns and their subsequent attack strings at the same time. Furthermore, we can detect vulnerabilities in regular expression patterns that support extended features, which are used in most modern web/mobile applications. Combining the tools significantly reduces the effort of separately executing the tools to obtain the desired results.

Further research is required to understand how the combination of SlowFuzz and REVEALER stacks up against algorithmic complexity attacks. While separate executions of SlowFuzz and REVEALER have given out promising results, their results might be overshadowed by combining SlowFuzz and REVEALER.

References

[1] Yinxi Liu, Mingxue Zhang, and Wei Meng: REVEALER: Detecting and Exploiting Regular Expression Denial-of-Service Vulnerabilities. Chinese University of Hong Kong.
[2] Theofilos Petsios Jason Zhao Angelos D.Keromytis Suman Jana. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities.
[3] L. G. Michael, J. Donohue, J. C. Davis, D. Lee, and F. Servant. Regexes are hard: Decision-making, difficulties, and risks in programming regular expressions. In Pro- ceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), San Diego, CA, Nov. 2019.
[4] Y. Shen, Y. Jiang, C. Xu, P. Yu, X. Ma, and J. Lu. ReScue: crafting regular expression DoS attacks. In Proceedings of the 33rd IEEE/ACM International Conference on Auto- mated Software Engineering (ASE), Montpellier, France, Sept. 2018.
[5] A. Rathnayake and H. Thielecke. Static analysis for regular expression exponential runtime via substructural logics (extended). arXiv preprint arXiv:1405.7058, 2014.
[6] V. Wüstholz, O. Olivo, M. J. Heule, and I. Dillig. Static detection of dos vulnerabilities in programs that use regular expressions. In Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Uppsala, Sweden, Apr. 2017.
[7] libFuzzer - a library for coverage-guided fuzz testing - LLVM 3.9 documentation.http://llvm.org/docs/LibFuzzer.html.
[8] V. Wüstholz, O. Olivo, M. J. Heule, and I. Dillig. Static detection of dos vulnerabilities in programs that use regular expressions. In Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Uppsala, Sweden, Apr. 2017.