Caught in the Web: DoS Vulnerabilities in Parsers for Structured Data[⋆]

Shawn Rasheed[1][0000−0001−7683−4296], Jens Dietrich[2][0000−0001−9019−6550], and Amjed Tahir[1][0000−0001−9454−1366]

[1] Massey University, Palmerston North, New Zealand
{s.rasheed,a.tahir}@massey.ac.nz
[2] Victoria University of Wellington, Wellington, New Zealand
jens.dietrich@vuw.ac.nz

Abstract. We study a class of denial-of-service (DoS) vulnerabilities that occur in parsing structured data. These vulnerabilities enable low bandwidth DoS attacks with input that causes algorithms to execute in disproportionately large time and / or space. We generalise the characteristics of these vulnerabilities, and frame them in terms of three aspects, TTT: (1) the Topology of composite data structures formed by the internal representation of parsed data, (2) the presence of recursive functions for the Traversal of the data structures and (3) the presence of a Trigger that enables an attacker to activate the traversal.
An analysis based on this abstraction was implemented for one target platform (Java), and in our study, we found that the impact of the results obtained with this method goes beyond Java. The inputs from our investigation revealed several similar vulnerabilities in programs written in other languages such as Rust and PHP. As a result we have reported 11 issues (of which seven have been accepted as issues), and obtained four CVEs for some of those issues in PDF, SVG and YAML libraries across different languages.

Keywords: DoS · Security · Vulnerabilities · Analysis.

1 Introduction

Denial-of-service (DoS) attacks based on algorithmic complexity have received increasing attention recently [29, 26].[3] Unlike classical DoS attacks based on flooding an application with network requests, or exploiting bugs that crash applications, algorithmic complexity-based DoS attacks target the exhaustion of computational resources such as CPU or memory, with small inputs that cause worst-case performance behaviour in a program [7]. Some of the more well-known attacks based on this class of vulnerabilities are regular expression DoS (*ReDoS*) [29] that target regular expression engines, and *HashDos* [7] that target hash functions.

[⋆] The work of the second author was supported by Oracle Labs, Australia

[3] https://blog.cloudflare.com/cloudflare-outage/ [Accessed 08-October-2020]

A complexity-related DoS vulnerability can be due to programs recursively traversing composite data structures. Tree and graph-like data structures that are composed of parts are common in programs, and when such structures are defined recursively, it is practical to define recursive operations over them. Such an operation can potentially run in exponential time or / and space if redundant traversals are not (or cannot be) controlled. These performance-related vulnerabilities can be exploited to carry out DoS attacks on systems. Serialisation,[4] which externalises a program's internal data structures to disk or for transmission over a network, and external format parsers that utilise these data structures present opportunities for DoS attacks based on these vulnerabilities [9].

Static and dynamic analysis techniques have been used for detecting performance-related bugs in programs. However, most existing approaches for detecting them are domain-specific - for instance, detection techniques for regular expression engines [33]. Fuzzing has been used to detect performance defects in programs. However, fuzzers are inefficient by nature with normal turnaround times in hours or more [16]. Constraint-based techniques such as symbolic execution are limited when it comes to producing complex inputs and there is little work on using them to detect performance bugs, especially for complex inputs [20, 23].

This work presents a characterisation of a class of vulnerabilities along with a novel approach to detect them. Our approach is based on modelling the vulnerabilities, implementing the analysis for the Java language and constructing payloads to verify the analysis reports. Finally, the constructed payloads are used to check if other libraries are vulnerable as well. We characterise some of the program structures that facilitate such an attack. This is broken down into three parts (we refer to these parts as the three T's):

1. **Topology:** a data structure that has a topology which allows the redundant execution of recursive code/methods.
2. **Traversal:** the presence of recursive methods that operate on the elements in the data structures identified in step one.
3. **Trigger:** an execution path, in a program, from an entry point method for the program, typically a method that loads and evaluates data, to a recursive method identified in step two.

We implemented this analysis for Java and then evaluated it on a set of 16 Java parser libraries for different data formats. The scope and impact of this study goes beyond the vulnerabilities found in the Java libraries as these libraries are used in numerous applications. We validated the vulnerabilities by constructing malicious inputs, and it turns out that some of these reveal vulnerabilities in libraries and applications written in other languages such as Rust and PostScript.

In our study, we found a total of *11* vulnerabilities: Four new vulnerabilities in Java libraries using the analysis, (i.e. Apache PDFBox, PDFxStream, Apache Batik and SnakeYAML), and seven vulnerabilities in non-Java libraries found

[4] Java serialisation, https://docs.oracle.com/javase/tutorial/jndi/objects/serial.html [Accessed 08-October-2020]

during the evaluation. All 11 issues were reported to the vendors (7 were accepted as security bugs) and four CVEs were obtained as a result. We have made the implementation and results publicly available for replication.[5]

2 Motivation

The vulnerabilities that we study are closely related to the well-known *billion laughs* attack on XML parsers [6] where the payload consists of an XML document with nested XML entities where each entity's definition contains references to the preceding definition. Parsing the document results in an output, which has a length exponential in the depth of the nesting, that causes the service to degrade or fail from memory exhaustion.

The basic idea of billion laughs can be ported to an attack on Java programs as shown by Coekaert's SerialDoS vulnerability [5]. In SerialDoS, shown in Listing 1.1, the equivalent of the nested entity references in an XML element is a serialised collection of nested sets.

Listing 1.1. SerialDoS payload construction

```
import java.util.*;
...
Set root = new HashSet();
Set s1 = root;
Set s2 = new HashSet();
for (int i = 0; i < depthN; i++) {
   Set child1 = new HashSet();
   Set child2 = new HashSet();
   child1.add("foo");
   s1.add(child1);
   s1.add(child2);
   s2.add(child1);
   s2.add(child2);
   s1 = child1;
   s2 = child2;
}

root.hashCode();
```

The problem occurs if `root.hashCode()` is invoked. The `hashCode` of `HashSet` is recursive, i.e., it is computed using the hash values of the elements of the respective sets. This leads to a computation that is exponential in `depthN`.

The first enabling property for this attack is the shape of the data structure, forming a network of cross-referencing parent-child relationships, where each child node is referenced by more than one (in this case: two) parent objects. The resulting object graph for the listing is shown in Fig. 1. The second ingredient is the recursive method that operates on this structure, `hashCode()` in this case.

Finally, there must be a way to parse the format to an internal representation with the same topology and trigger the traversal over it. The fact that the method can be reached from the entry point of a program can be statically determined by examining direct paths from the entry point method to the target method in the call graph.

[5] https://bitbucket.org/unshorn/ciwstudy/

In the case of SerialDoS, the trigger is the deserialisation API. Serialisation is a common attack surface for Java applications as demonstrated by Frohoff et al. [2, 10]. The deserialisation of the `HashSet` instances encountered in the stream will then invoke `hashCode()` via a call graph chain from `readObject()`, and the malicious computation is activated.

Redundant traversals can be solved using dynamic programming in some cases. However, a solution is not available in the case of SerialDoS where it is not a programming defect that gives rise to the vulnerability. In SerialDoS, each method invocation happens within another context (i.e., state of the stack), and for a programmer to cache intermediate results would require knowledge about the state of the heap and stack during each invocation.

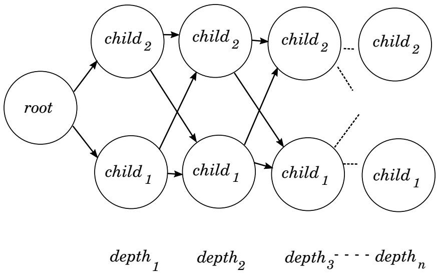

Fig. 1. Many-to-many topology in object graphs – each vertex represents an object and an edge represents a reference. Note that parents have two children, but children also have two parents. The graph is abstracted in the sense that intermediate references to elements in `java.util.HashSet` are not shown.

3 Characteristics of the Vulnerability

In this section, we discuss the three defining characteristics, TTT, needed to craft DoS attacks based on vulnerabilities in code recursively traversing data structures: *topologies*, *traversals* and *triggers*.

3.1 Topologies

In Java-like languages, an instance of a data structure forms a graph of objects as depicted in Fig. 1. An object, *parent*, references $child_1$ if $child_1$ is the value of a field of *parent*. Often, references are indirect via intermediate objects, in particular arrays or collections. Given an object graph, we are particularly interested in subgraphs formed by objects of some type T, where these objects have more than one predecessor and/or successor of type T. These structures can be described as following a many-to-many pattern.[6] Examples of this are common in the Java collection library, where, for instance, lists can be elements of multiple other lists.

A pattern that is very widely used, and conceptually similar to many-to-many, is *composite* [11]. A composite has containers with elements that are either containers as well, or leaf nodes. Usually, there are dedicated container and child types subtyping a more abstract component type, but variations of this pattern are common.[7]

Whether a composite can form a many-to-many graph depends on how the parent-child relationships are represented and controlled. Often it is an implicit precondition for an operation with a composite as an input, that at most, the composite has a single reference to any of its children. This is often enforced by an explicit parent reference in the children, and an API that maintains the consistency of the parent-child and child-parent references.[8]

3.2 Traversals

A many-to-many pattern in an object graph describes the data structure that can be exploited in SerialDoS-style attacks. To actually launch an attack, a function that operates recursively on this structure must be present. Regarding these functions, or methods as they are called in object-oriented languages, there are two aspects to consider. Firstly, recursion can be simply described by the presence of strongly-connected components in the call graph.

Secondly, the recursive method needs to traverse the many-to-many object graph. Consider such a method m with formal parameters $m.this, m.arg_1, m.arg_n$. Then, there must be a call graph chain linking m to itself. Additionally, at this invocation there is a parameter that points to a child of an object that the respective parameter points to at the previous invocation.

This captures several common *traversal patterns* found in real world programs. The most obvious one is direct recursion, where the many-to-many elements are always referenced by `this`.

There are more complicated traversal patterns though. For instance, it is possible to implement traversals using static methods in some class, where the traversed data structure is only passed as a parameter. This can be used to program traversals implemented outside the data structure being traversed. A more standardised way to achieve this is using the *visitor* pattern [11], which factors out the operation to be performed on a data structure from its implementation.

[6] Many-to-many relationships, in the database community, describe one type of cardinality of relationships between two entities.

[7] For instance, in `java.io.File`, state is used to determine whether an element is a container (a directory) or a child (a file). This can be considered a case of a composite that uses structural instead of nominal typing.

[8] A good example for this is how the parent reference is maintained in the `java.awt.Container.add*` methods which add a child component to the visual component hierarchy.

Note that the component of traversals we have discussed so far concerns traversing the graph depth-wise. Another aspect is how a method traverses the children of each parent. In our example, this is accomplished through the for-loop which iterates over the children in `elements`. A generalised technique is to use the iterator pattern [11], which abstracts the traversal of different implementations of collection types.

3.3 Triggers

The presence of the topology and the traversals itself is not sufficient to launch an attack. It is also necessary to create objects that instantiate the many-to-many pattern, and to trigger the traversal. This requires an external data structure to be translated to a vulnerable internal representation of the object graph in a data format, interpreted or otherwise processed, and in this process the traversal is activated. The activation refers to an invocation chain from a method that is called by the library or the framework when the data is processed, to the actual traversal method (e.g., deserialisation in SerialDoS). In general, a trigger is a method in the public API of a library that is invoked when data is processed (either by a client program, or by the library itself), and that leads to the invocation of the traversal (method).

4 Modelling the Analysis

In this section, we describe how we model the topology, traversal and trigger pattern for our analysis. We formalise the analysis in graph-theoretic terms using three graphs that model different aspects of program behaviour. These definitions are language-agnostic, and are applicable to languages with common object-oriented language features. We occasionally refer to Java for illustrative purposes.

4.1 Preliminaries

The Type Graph A type graph models classes and their relationships in a program written in an object-oriented programming language, formally defined as follows:

Definition 1. *The* ***type graph*** *of a program is a directed labelled graph* $\langle T, E\rangle$ *where*

- T *is a set of vertices representing classes and (nested) arrays of classes that occur in the program, we represent vertices using their class name, with the suffix* $[\,]$ *for array types as usual. For the sake of brevity, we identify those names with the respective vertices from now on.*
- $E \subseteq T \times T$ *are the graph edges. We consider edges of two kinds, i.e.,* $E = E_{subtype} \cup E_{assoc}$ *, representing subtype and association relationships, respectively.*

Subtype edges between vertices represent subtype relationships. The particular rules differ between languages, for Java, they are defined in [13, sect 4.10]. We use the simplified notion "`B` is subtype of `A`" to say that there is a path consisting of subtype edges linking `B` to `A`. That is, the edge $(B \rightarrow A) \in E_{subtype}$. If $C \in T$ is either representing an array, or a class that is a subtype of a container type (such as Java's `java.util.Collection`), then we call C a container type. If a container type is an array `A[]` then `A` is called the component type of the container. If the language supports the declaration of component types for container types, using some mechanism like Java's generics [13, sect. 8.1.2], `C<A>`, then `A` is called the component type of `C`. Association edges represent the relationships between container types and their respective component types.

The Points-To Graph A points-to graph models the memory during program executions. There are several versions of points-to analysis, our representation is used in an Andersen-style, field-sensitive analysis [30]. In a points-to graph, object abstractions and variables are represented by vertices, and allocations, assignments, field or array stores and loads are represented by edges which define the flow of values in the program being represented.

Definition 2. *The **points-to graph** of a program is a directed labelled bipartite graph* $\langle O, V, E_{alloc}, E_{assign}, E_{load}, E_{store} \rangle$ *where*

- V *is the set of vertices representing the variables in the program*
- O *is the set vertices representing object allocation sites in the program*
- $E_{alloc} \subseteq O \times V$ *is a set of allocation edges, modelling memory allocation*
- $E_{assign} \subseteq V \times V$*, a set of assignment edges modelling variable assignment*
- $E_{load} \subseteq V \times V$*, a set of field load edges modelling field loads, labelled with the respective field name*
- $E_{store} \subseteq V \times V$*, a set of field store edges modelling field stores, labelled with the respective field name*

Array access can be modelled similarly to field access. We omit details for the sake of brevity here.

Given a points-to graph, the objective of a points-to analysis is to infer additional *flowsTo* edges $E_{flowsTo} \subseteq O \times V$ describing the relationship between abstract values and variables pointing to them.[9] This is a computational complex problem, usually solved by computing CFL-reachability via a fixpoint algorithm [28, 8].

One of the main uses of the points-to graph is alias analysis. Two variable vertices v_1, v_2 *alias* if there is an object vertex $o \in O$ and paths consisting of $flowsTo$ edges from o to both variable vertices v_1 and v_2. Aliasing means that both variables can point to the same memory location. Furthermore, this can be used to define a *heap access path* between variables. A heap access path between v_1 and v_2 consists of a sequence of load edges where the destination (sink) of an edge in the sequence aliases with the source of the next edge, v_1 is the source

[9] Sometimes, the reverse *points-to* edges are inferred

of the first and v_2 the destination of the last edge in the sequence. This models (nested) field access in a programming language, i.e. statements like `foo.f.g`. By accounting for aliases, this also covers field access with intermediate variables, in programs like `x = foo.f; x.g;`.

The Call Graph A call graph statically models the interprocedural calling (invocation) behaviour of a program. Methods are represented by vertices and interprocedural calls by edges.

Definition 3. *The **call graph** of a program is a directed graph $\langle M, I\rangle$ where*

- *M is the set of methods in the program*
- *$I \subseteq M \times M$ is a set of edges, $(m, n) \in I$ means that method m has a call site with an invocation of n.*

A call graph can be constructed by analysing invocation instructions found in program code. To model runtime behaviour in languages with dynamic dispatch, additional edges must be inferred (devirtualisation). There are various algorithms available for this purpose differing in their precision and efficiency (e.g. CHA, VTA) [30]. The more precise methods require points-to information to determine the type of the objects a receiver points to in a method invocation, a process often referred to as call graph construction on-the-fly.

Cross-Referencing Graphs The three models defined above are widely used in static program analysis. In practice, they are often combined. There are certain relationships between these models we will exploit in our analysis.

Firstly, for a given method m, the variables in the points-to graph include the return value and the parameters of this method. We denote the parameters, including the receiver of an invocation (in Java, the `this` reference), as $param(m) \subseteq V$. Secondly, call graph vertices can be associated with type graph vertices via the types that define those methods. We refer to those types as the owner of a method m, $owner(m) \in T$. Finally, allocation vertices $o \in O$ in the points-to graph can be associated with the types they instantiate, $type(o) \in T$.

4.2 Analysis Specification

Topologies Given a type graph $\langle T, E\rangle$, we describe an instance of *composite* as a mapping between the two roles in the composite design pattern [11] and actual types that occur in the program.[10]

Definition 4. *Given a type graph $TG = \langle T, E\rangle$, a composite is a mapping $\{cont, comp\} \rightarrow TG$ such that the following two conditions are satisfied:*

- *$(composite(cont), composite(comp)) \in E_{assoc}$*
- *$(composite(cont), composite(comp)) \in E_{subtype}$*

[10] The *cont* role corresponds to the *Container* role in the design pattern, whereas the *comp* roles corresponds to the *Component* role. We do not consider a particular leaf type.

Traversals We define a traversal as the presence of a recursive invocation in the call graph and the flow of an object from a field of a composite to an argument of the recursive call.

Definition 5. *Given the type graph $TG = \langle T, E \rangle$, the call graph $CG = \langle M, I \rangle$ and the points-to graph $PG = \langle V, O, E_{alloc}, E_{assign}, E_{load}, E_{store} \rangle$, a traversal is a method $m \in CG$ such that:*

- *$m \in M$ is recursive in CG, i.e. CG contains a path connecting m to itself*
- *there is a composite c and $c(comp) \in param(m)$*
- *there is a heap access path in PG from some parameter of $v \in param(m)$ to the respective argument in the recursive call site for m*

Triggers We define a trigger as the presence of a method that instantiates a *composite* through a chain of method invocations. Once the composite is instantiated it would also trigger the recursive traversal with the *composite* as an argument.

Definition 6. *Given a call graph $CG = \langle M, I \rangle$ and the points-to graph $PG = \langle V, O, E_{alloc}, E_{assign}, E_{load}, E_{store} \rangle$, a trigger is a method $trigger \in M$ in the call graph that is reachable from a program entry point $ep \in M$. There must also be a path in the CG from the trigger to a traversal method with the instantiated composite as an argument in PG.*

5 Experimental Setup and Evaluation

5.1 Approach

The approach focuses on first running the static analysis to detect instances of the TTT pattern in Java parser libraries, then using the analysis results to confirm whether we can construct payloads to exploit vulnerabilities, and finally using the payload for non-Java parsers (e.g., C, Python, Rust etc...) to check if these parsers are also vulnerable. The payloads were constructed manually using format specifications, source code indicated by analysis results. The effort required to construct payloads varied across different libraries, and this process is discussed in more detail in section 5.5.

5.2 Implementation

The analysis was implemented as an extension of Doop [1], a state-of-the-art static analysis framework for Java, which encodes static analyses declaratively in the Datalog [4] language. Datalog programs are a natural way to express the graph-based algorithms [35] used in the specification of our analysis. Datalog-based formulations of static analyses have been used successfully in bug and vulnerability detection [14, 18, 27]. The underlying analyses in Doop compute points-to information and call graphs from an input program, and they can also be used to obtain the type graph.

5.3 Libraries for Analysis

We evaluated the analysis on a set of 16 widely used Java parser libraries for different data formats (Table 1). These popular libraries are known to parse external data formats, and are therefore prone to the vulnerabilities we study. These libraries process data used in messaging, object serialisation and document representation, represented in various text and binary formats. We covered libraries for parsing or processing XML, JSON and YAML, PDF and external DSLs. Table 1 also contains usage data showing how many Maven artefacts depend on those libraries. This provides some indication of the impact vulnerabilities in these libraries have based on usage statistics.[11]

Table 1. Java Libraries for Analysis

Library	**Input Format**	**Version**	**Usage**
batik	SVG	1.1	115
gson	JSON	2.8.5	11,900
jackson	JSON	2.9.8	6,829
jettison	JSON	1.4.0	753
jfxrt	FXML	1.8	N/A
mongo	BSON	3.9.1	1,048
mvel2	MVEL2	2.4.3	395
ognl	OGNL	3.2.10	339
pdfbox	PDF	2.0.12	403
pdfxstream	PDF	3.7.0	N/A
protobuf	Protocol	3.6.1	2,407
sanselan	Images	0.97	52
snakeyaml	YAML	1.23	1,962
stringtemplate	StringTemplate	3.2	270
xbean	SOAP/XMLBean	3.0.2	632
xstream	XStream	1.4.11.1	1,711

After analysing these 16 libraries, we proceeded to evaluate whether libraries for other languages, shown in Table 2 were vulnerable using the payloads constructed. These libraries were selected based on availability and popularity. librsvg is used in the GNOME desktop,[12] Ghostscript is widely deployed and used for processing PDFs, a popular vector-based illustration software - Inkscape uses librsvg and cairo.[13] Inkscape[14] is also used by ImageMagick for SVG processing. We looked at solutions for sanitising SVG files, and found that svg-sanitizer is the most widely used (e.g. WordPress, drupal).

[11] Maven usage statistics (obtained on 12 Feb. 2020).

[12] https://www.gnome.org

[13] https://cairosvg.org/

[14] https://inkscape.org/

Table 2. Non-Java libraries investigated

Library	**Language**	**Input Format**	**Version**
Qt	C++	SVG	5.14.1
librsvg	Rust	SVG	2.46
PDFtk	GCJ	PDF	2.0.2
qpdf	C++	PDF	9.1.1
PDFium	C++	PDF	N/A
ghostscript	PostScript	PDF	9.25
svg-sanitizer	PHP	SVG	0.13.2
resvg	Rust	SVG	0.8.0
cairosvg	Python	SVG	2.4.2

5.4 Triggers or Entry Points

Identifying a trigger is a manual step that requires domain knowledge of the library under analysis. For image formats this could be the rasterisation/conversion process that would require traversals of the structure. Some libraries have command line interfaces which initiate calls to the trigger methods. In the case where we analysed libraries without command line interfaces, a driver was required as an entry point for the input program for the analysis. We have written custom drivers for libraries that are not bundled with a command line interface. The driver provides an entry point as well as a facility to interact with the library's API. In the case of SnakeYAML, the driver consists of statements to instantiate the parser and load a file. Only MVEL2, PDFBox, Batik and PDFxStream come with built-in command line interfaces and did not require custom drivers.

5.5 Evaluation

Static Analysis The experiments were performed on an Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz with 64GB of RAM on Linux Ubuntu 18.04.3. Doop was run using the Java 8 platform as implemented in Oracle's version 8 of the JDK. We used a context-insensitive analysis. the following options for the analysis:

- analysis: `-a context-insensitive`
- main class: `-main` option was used with the driver class as an argument.

For each project, we extracted facts from the input library, and then executed Doop with custom rules to compute:

- Composites (i.e., facts instantiating the Composite rule).
- Recursive methods.
- Heap flows to refine the list of recursive methods, i.e. an object that is of composite type must flow to the parameter of a recursive method.
- Methods with heap flows from the previous step, and that are reachable via the entry point.

Manual Evaluation of Analysis Results At the end of the static analysis we find a set of candidate instances, i.e. bindings of the concepts used in TTT to concrete artefacts within the program under analysis. However, these may contain false positives as the malicious computation is effectively prevented by some program logic. While we cannot accurately eliminate false positives, we conducted a manual step to identify true positives, by constructing payloads that expose the respective vulnerability. This consisted of inspection of the program's source code, debugging and reviewing specifications against the implementation for the particular parser library.

6 Results and Discussion

Table 3 shows a summary of the outcomes of the experiment, including the static analysis run times for the 16 libraries. Methods and composites from the library and their dependencies are shown. The **Topology** column lists all composite types in the library. The **Methods (Composite)** column lists only those methods that have a composite type as a parameter. From these methods, the **Traversal** column lists methods that have a value flow within the composite field to the recursive callsite. The **Triggered Traversals** column lists reachable traversals for the identified topologies. In the following sections, we discuss some of the vulnerabilities detected in more detail.

Table 3. Overview of Experiments (Composites and Direct Recursion)

Library	Format	Time (sec)	Topology	Methods (Composite)	Traversal	Triggered Traversals
batik	SVG	505	430	595	34	34
gson	JSON	300	27	25	1	1
jackson	XML	404	126	167	10	10
jettison	JSON/XML	296	19	8	1	1
jfxrt	XML	784	0	0	0	0
mongo	BSON	433	275	224	0	0
mvel2	MVEL	173	93	135	4	2
ognl	OGNL	317	29	64	4	4
pdfbox	PDF	703	480	247	11	11
pdfxstream	PDF	334	115	118	3	3
protobuf	Protobuf	383	199	202	5	5
sanselan	Image	307	35	6	0	0
snakeyaml	YAML	307	30	13	3	3
stringtemplate	Template	306	32	18	0	0
xbean	XML	492	363	230	0	0
xstream	XML	331	120	100	1	1

6.1 PDF Vulnerabilities

The analysis detected 11 triggered traversals that recurse on a parameter in the PDFBox library. From these results, a vulnerability was confirmed in Apache PDFBox, the most used Java PDF library in the Maven repository.[15]

A PDF document's format, Carousel Object Structure (COS), is described in the PDF Reference [24]. It supports basic types such as booleans, integers, real numbers, strings, names and more crucially, arrays and dictionaries. The particular composite topology in the library consists of `COSDictionary` as the container and `COSBase` as the component where the children are stored in an object that implements the `Map` interface. The recursive method that traverses this structure is `checkPagesDictionary(COSDictionary pagesDict)` defined in `org.apache.pdfbox.pdfparser.COSParser`, which is invoked when the PDF file is parsed. The only constraint in the path condition from the entry into the method to the recursive call is the presence of child objects that are of the same type as the passed parameter.

Manual inspection of the source code and the PDF specification [24] revealed that the root of a PDF document, the catalog, points to a dictionary referred to as a *Page Tree*, which can in turn refer to another *Page Tree*. This structure parses to a `COSDictionary` composite and we can craft a PDF document that parses into an object graph with the many-to-many pattern.

Passing the crafted PDF to the application revealed that it can result in attacks on responsiveness and disk space as the application can also be used to convert the pages in a PDF to disk as images. The issue was reported and it has been accepted with the identifier CVE-2018-11797.

The same PDF document was used to confirm the vulnerability in PDFxStream (CVE-2019-17063), and it also revealed the same vulnerability in PDFtk[16], the PDF toolkit. They both use a `HashMap` to store the COS structure, which in principle makes a DoS attack possible.

We also tested the PDF document on Ghostscript [12], a PostScript and PDF interpreter. Using the crafted PDF as an input to Ghostscript resulted in DoS. This bug was accepted as a security vulnerability (CVE-2018-19478). The PDF parser in Ghostscript is implemented in PostScript and the traversal of the COS was for an entirely different purpose when compared to the previous cases, which was to detect cycles in the *Page Tree* that had caused a security vulnerability in Ghostscript. This suggests that traversals of the form that we have studied occur across multiple languages.

6.2 Scalable Vector Graphics (SVG) Vulnerability

SVG [31] is an XML-based format for two-dimensional graphics supported by web browsers and is used in illustration programs. SVG is processed by the Batik library. The analysis reported 34 triggered traversals for the library. One

[15] https://mvnrepository.com/

[16] https://www.pdflabs.com/tools/pdftk-the-pdf-toolkit/ [Accessed 08-October-2020]

particular composite topology in the library consists of `Node` as the container with children or parents of the same type. The recursive method that traverses this structure is `String getCascadedXMLBase (Node node)` defined in `org.apache.batik.anim.dom.SVGOMElement`, which is invoked when the SVG file is parsed. The only constraint in the path condition from the entry into the method to the recursive call is the presence of child objects that are of the same node type as the passed parameter (i.e. XML element nodes).

Any SVG graphics element is potentially a template object that can be reused (i.e., instanced) in the SVG document via a `<use>` element. The `<use>` element references another element and indicates that the graphical contents of that element is included/drawn at that given point in the document. The `<g>` element can be used to specify a grouped container of elements. The `<g>` element in conjunction with the `<use>` element can be used to construct a nested structure to trigger the detected vulnerability. The `<use>` element can also be used to construct the SVG version for SerialDoS as shown in Listing 1.2. We based our SVG file on this ability to nest references using `<g>` and `<use>` elements. There is an additional way to reference elements, as shown in Listing 1.3, which uses the pattern tag and its fill attribute set to a `url` function containing the reference id for an element in the document.

Listing 1.2. Nested References in SVG

```
<g id="t0a">
<use xlink:href="#t1a"/>
<use xlink:href="#t1b"/>
</g>

<g id="t0b">
<use xlink:href="#t1a"/>
<use xlink:href="#t1b"/>
</g>
```

Listing 1.3. References with use() function in SVG

```
<pattern id="h" ... >
      <rect  fill="url(#g)" stroke="green" />
```

The same SVG document was used to verify vulnerabilities in web browsers, and a core Linux SVG rendering library (librsvg[17]). The issue was reported for librsvg and fixed by the vendor (CVE-2019-20446). We found the crafted file to impact all tested browsers (e.g. Mozilla Firefox (version 73.0), Google Chrome (Version 77.0.3865.120, Official Build) by excessively consuming resources (memory and CPU) for Firefox and crashing the active browser tab in Chrome. This can be used by malicious parties to craft client-DoS for websites that allow links to SVG code in user input (e.g. Markdown with links to external SVG files in user comments). We confirmed this observation for the StackOverflow[18] Q&A platform, GitLab[19] and GitHub[20] issue trackers. The impact on these services

[17] https://wiki.gnome.org/action/show/Projects/LibRsvg [Accessed 08-October-2020]

[18] https://stackoverflow.com

[19] https://gitlab.com

[20] https://github.com

is that they can render the page inaccessible to users if it has malicious SVG content.

We also considered svg-sanitizer[21], which performs server-side sanitisation of SVG content (used in Drupal and WordPress as a plugin). On passing the crafted SVG file as input, svg-sanitizer, entered a non-terminating computation which make services using the plugin susceptible to DoS attacks. This issue was reported to the developer and it was fixed by adding a check to limit levels of `use` recursion during SVG sanitisation.

6.3 YAML Vulnerability

YAML is a popular and widely used (human readable) serialisation language for data interchange and application configuration. It supports primitives and common data structures such as maps and lists [34]. We looked at the SnakeYAML library in Java and the analysis reported three triggered traversals, one of which involves the composite `MappingNode` with a list of `NodeTuple` as children that can potentially have the same type as the parent.

The code is the implementation of the `<<` merge key feature in YAML, which is used to indicate that all the keys of one or more specified maps should be inserted into the current map. If the value associated with the key is a single map, each of its key/value pairs is inserted into the current map, unless the key already exists in it. If the value associated with the merge key is a sequence, then this sequence is expected to contain multiple maps and each of these are merged in order. Listing 1.4 shows the use of YAML merge as well as the use of YAML aliases in constructing SerialDoS type inputs, which were detected as vulnerabilities by the analysis.

Listing 1.4. Merging map keys in YAML

```
? - &t2a
    - &t3a [lol]
    - &t3b [lol]
  - &t2b
    - *t3a
    - *t3b
: value
--
{ << { << { key: value} } }
```

We created a YAML file with nested merges and nested lists with aliases forming the topology, and passed the file as input to our SnakeYAML driver to confirm that it crashed from stack exhaustion for the nested merges case, and entered a long-running computation for nested lists. Consequently, this issue has been reported to the maintainer.

6.4 Newly Discovered Security Vulnerabilities

Following the guidelines for responsible disclosure, we have reported all vulnerabilities to the libraries' developers/maintainers. We provide a timeline and the statuses of each of these vulnerabilities below.

[21] https://github.com/darylldoyle/svg-sanitizer [Accessed 08-October-2020]

Table 4. Status of reported bugs and vulnerabilities status.

Library	Version	Status	Fixed date
PDFBox	2.0.12	CVE-2018-11797	6-Oct-18
PDFxStream	3.6.0	CVE-2019-17063	27-Feb-19
PDFTk	2.02	Pending	
GhostScript	9.25	CVE-2018-19478	20-Nov-2018
Svg-sanitizer	0.13.0	CVE requested (pending)	20-Jan-20
Batik	1.11	Won't fix	-
Firefox	69	Duplicate	-
Drupal	6.x-8.x	-	25-June-20
Snakeyaml	1.23	Won't fix	
Qtsvg	5.14.1	Bug	29-Feb-20
Librvg	2.46.2	CVE-2019-20446	15-Oct-19

6.5 Threats to Validity

Manual confirmation of the vulnerabilities reported by the tool poses a threat to the validity of the evaluation. However, for the most likely candidates, we were able to construct payloads and confirm that the reports are actual bugs. Even though a hand-selected set of projects was used in the evaluation, the generality of the model and the discovery of related bugs in other libraries are encouraging.

7 Related Work

7.1 Detecting Algorithmic Complexity vulnerabilities

Wuestholz et al. [33] discuss an approach to statically detect DoS vulnerabilities in programs that use regular expressions. The analysis has multiple stages and is conceptually similar to the analysis proposed in this paper: they first build a model to detect vulnerable structures (by reasoning about the worst-case complexity of NFAs), and then devise a separate (taint) analysis to establish whether a vulnerable regular expression can be matched against an input string. The tool they have developed, Rexploiter, finds 41 exploitable security vulnerabilities in Java web applications. Holland et al. [15] propose a hybrid approach to detect algorithmic complexity vulnerabilities. In a static pre analysis step, they use a loop call graph to detect nested loop structures that are susceptible to algorithmic complexity vulnerabilities. The first step is similar to our approach, but uses a different model. Our approach is based on the presence of higher level data structures and recursive methods which then implicitly create the nested traversals.

7.2 Traversals / Performance bugs

The detection of performance bugs and in particular redundant traversal is a problem related to DoS vulnerabilities. Olivo et al. [21] study redundant traversal performance bugs in Java code, limited to traversals in non-recursive functions, and a static analysis, Clarity, to detect them. Burnim et al. [3] present WISE, automated test generation for detecting worst-case complexity in programs. WISE uses symbolic test generation. Jiayi et al. [32] describe Singularity, another input generation technique for detecting worst-case performance bugs in Java programs. Singularity uses a greybox fuzzing technique that looks for critical input patterns modelled as recurrent computation graphs (RCGs). Their technique reveals performance and DoS-related bugs in real world programs. Other fuzzing approaches include SlowFuzz and PerfFuzz [17, 25]. Nistpor et al. [19] propose Toddler, an example of dynamic analysis to detect performance bugs. Toddler instruments loops and read instructions, and uses the data collected using inserted code to detect similar memory-access patterns. Padhye and Sen [22] describe Travioli, a dynamic analysis technique for detecting data-structure traversals. It is also based on instrumenting code in order to harvest trace data, from which the analysis model is built. The purpose is similar to what we try to achieve in this paper with the topologies and traversal steps of our model, however this being a dynamic model, it has different tradeoffs between precision and recall, and its quality depends on the existence of drivers (such as unit tests) that exercise a large part of the program under analysis.

8 Conclusion

We presented an approach to classify and detect a class of DoS vulnerabilities in parsing data structures. We evaluated this approach on a set of 16 Java parser libraries with a Datalog-based formulation of a static analysis using the Doop analysis framework for Java. The study revealed four new vulnerabilities in widely used Java PDF, SVG and YAML libraries. A further evaluation also revealed seven more vulnerabilities in parser libraries for Rust, PHP, C++ and PostScript. Out of these reports, we have obtained four CVEs and reported a total of 11 security issues to vendors (7 of which have been accepted). The results confirm that a lightweight static analysis can be useful in uncovering vulnerabilities that belong to this class. Possible directions for future work include using micro-fuzzing to fuzz the recursive functions reported by the static analysis for more precise results, and using constraint-based approaches to more precisely identify the topology and traversal patterns reported by the analysis.

References

1. Bravenboer, M., Smaragdakis, Y.: Strictly Declarative Specification of Sophisticated Points-to Analyses. In: Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications. p. 243–262. OOPSLA '09, Association for Computing Machinery, New York, NY, USA (2009). https://doi.org/10.1145/1640089.1640108
2. Breen, S.: What Do WebLogic, WebSphere, JBoss, Jenkins, OpenNMS, and Your Application Have in Common? This Vulnerability. https://goo.gl/cx7X4D (2015), [Online; accessed 08-October-2020]

3. Burnim, J., Juvekar, S., Sen, K.: WISE: Automated Test Generation for Worst-case Complexity. In: Proc. ICSE'09. IEEE (2009)
4. Ceri, S., Gottlob, G., Tanca, L.: What you always wanted to know about Datalog (and never dared to ask). IEEE TKDE **1**(1) (1989)
5. Coekaerts, W.: SerialDOS. https://gist.github.com/coekie/a27cc406fc9f3dc7a70d (2015), [Online; accessed 08-October-2020]
6. CVE-2003-1564 (Billion Laughs). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2003-1564 (2003), [Online; accessed 14-January-2020]
7. Crosby, S.A., Wallach, D.S.: Denial of Service via Algorithmic Complexity Attacks. In: Proc. USENIX Security'03. USENIX Association (2003)
8. Dietrich, J., Hollingum, N., Scholz, B.: Giga-scale exhaustive points-to analysis for Java in under a minute. In: Proc. OOPSLA'15. ACM (2015)
9. Dietrich, J., Jezek, K., Rasheed, S., Tahir, A., Potanin, A.: Evil Pickles: DoS Attacks Based on Object-Graph Engineering. In: Proc. ECOOP'17 (2017)
10. Frohoff, C., Lawrence, G.: Marshalling Pickles. http://frohoff.github.io/appseccali-marshalling-pickles/ (2015), [Online; accessed 08-October-2020]
11. Gamma, E., Vlissides, J., Johnson, R., Helm, R.: Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley (1994)
12. GhostScript, An interpreter for the PostScript language and for PDF. https://www.ghostscript.com/ (2019), [Online; accessed 14-January-2020]
13. Gosling, J., Joy, B., Steele, G., Brache, G., Buckley, A.: The Java® Language Specification Java SE 8 Edition. https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf (2015), [Online; accessed 08-October-2020]
14. Grech, N., Smaragdakis, Y.: P/taint: Unified points-to and taint analysis. In: Proc. OOPSLA'17. ACM (2017)
15. Holland, B., Santhanam, G.R., Awadhutkar, P., Kothari, S.: Statically-informed dynamic analysis tools to detect algorithmic complexity vulnerabilities. In: Proc. SCAM'16. IEEE (2016)
16. Klees, G., Ruef, A., Cooper, B., Wei, S., Hicks, M.: Evaluating fuzz testing. In: Proc. CCS'18. ACM (2018)
17. Lemieux, C., Padhye, R., Sen, K., Song, D.: PerfFuzz: Automatically Generating Pathological Inputs. In: Proc. ISSTA'18. ACM (2018)
18. Livshits, V.B., Lam, M.S.: Finding Security Vulnerabilities in Java Applications with Static Analysis. In: Proc. USENIX Security'14. USENIX Association (2005)
19. Nistor, A., Song, L., Marinov, D., Lu, S.: Toddler: Detecting performance problems via similar memory-access patterns. In: Proc. ICSE'13. IEEE (2013)
20. Noller, Y., Kersten, R., Păsăreanu, C.S.: Badger: Complexity analysis with fuzzing and symbolic execution. In: Proc ISSTA'18. ACM (2018)
21. Olivo, O., Dillig, I., Lin, C.: Static detection of asymptotic performance bugs in collection traversals. In: Proc. PLDI'15. ACM (2015)
22. Padhye, R., Sen, K.: Travioli: a dynamic analysis for detecting data-structure traversals. In: Proc. ICSE;17. IEEE (2017)
23. Păsăreanu, C.S., Kersten, R., Luckow, K., Phan, Q.S.: Symbolic execution and recent applications to worst-case execution, load testing, and security analysis. In: Advances in Computers, vol. 113. Elsevier (2019)
24. PDF Reference 6th edition. https://www.adobe.com/content/dam/acom/en/devnet/-pdf/pdf_reference_archive/pdf_reference_1-7.pdf (2006), [Online; accessed 14-January-2020]

25. Petsios, T., Zhao, J., Keromytis, A.D., Jana, S.: SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. In: Proc. CCS'17. ACM (2017)
26. Rasheed, S., Dietrich, J., Tahir, A.: Laughter in the Wild: A Study into DoS Vulnerabilities in YAML Libraries. In: Proc. TrustCom'19). IEEE (2019)
27. Scholz, B., Jordan, H., Subotić, P., Westmann, T.: On Fast Large-Scale Program Analysis in Datalog. In: Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016. ACM (2016)
28. Sridharan, M., Gopan, D., Shan, L., Bodík, R.: Demand-driven points-to analysis for Java. In: Proc. OOPSLA'05. ACM (2005)
29. Staicu, C.A., Pradel, M.: Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers. In: Proc. USENIX Security'18. USENIX Association (2018)
30. Sundaresan, V., Hendren, L., Razafimahefa, C., Vallée-Rai, R., Lam, P., Gagnon, E., Godin, C.: Practical virtual method call resolution for java. In: Proc. OOPSLA'00. ACM (2000)
31. Scalable Vector Graphics (SVG) 1.1 (2nd edition). https://www.w3.org/TR/SVG11/REC-SVG11-20110816.pdf (2011), [Online; accessed 14-January-2020]
32. Wei, J., Chen, J., Feng, Y., Ferles, K., Dillig, I.: Singularity: Pattern fuzzing for worst case complexity. In: Proc. ESEC/FSE'18. ACM (2018)
33. Wüstholz, V., Olivo, O., Heule, M.J., Dillig, I.: Static Detection of DoS Vulnerabilities in Programs That Use Regular Expressions. In: Proc. TACAS'17. Springer (2017)
34. YAML Ain't Markup Language (YAML) Version 1.2 . https://yaml.org/spec/1.2/spec.html (2019), [Online; accessed 08-October-2020]
35. Yannakakis, M.: Graph-theoretic methods in database theory. In: Proc. PODS'90. ACM (1990)