JChainz: Automatic Detection of Deserialization Vulnerabilities for the Java Language

Luca Buccioli[1(✉)], Stefano Cristalli[3], Edoardo Vignati[1], Lorenzo Nava[3], Daniele Badagliacca[1], Danilo Bruschi[1], Long Lu[2], and Andrea Lanzi[1]

[1] University of Milan, Milan, Italy
luca.buccioli@unimi.it
[2] Northeastern University, Boston, USA
[3] Security Pattern Inc., Milan, Italy

Abstract. In the last decade, we have seen the proliferation of code-reuse attacks that rely on deserialization of untrusted data in the context of web applications. The impact of these attacks is really important since they can be used for exposing private information of the users.

In this paper, we design a tool for automatic discovery of deserialization vulnerabilities for the Java language. Our purpose is to devise an automatic methodology that use a set of program analysis techniques and is able to output a deserialization attack chain. We test our techniques against common Java libraries used in web technology. The execution of our tool on such a dataset was able to validate the attack chains for the majority of already known vulnerabilities, and it was also able to discover multiple novel chains that represent new types of attack vectors.

1 Introduction

In the last decade, we see a propagation of code-reuse attacks in the context of web applications [4,11,14]. The impact of these attacks is important, since such vulnerabilities can be used for exposing several pieces of private information like credit card numbers, social security numbers of the common users. One example of this attack is direct to the agency Equifax, which exposes information on 143 million of US users. This attack exploits a well-known vulnerability named untrusted data deserialization in the web application context. In particular the insecure deserialization in the Apache Struts framework within a Java web application ends up in remote code execution (RCE) on Equifax web servers. The attack exploited the XML serialization data objects into textual strings and inject malicious XML payloads into Struts servers during the deserialization process. Such attacks show the need to systematically face code-reuse attack problems at research level.

More precisely to exploit this type of vulnerability, an attacker should create a custom instance of a chosen serializable class which redefines the readObject

G. Lenzini and W. Meng (Eds.): STM 2022, LNCS 13867, pp. 136–155, 2023.
https://doi.org/10.1007/978-3-031-29504-1_8

method. The object is then serialized and send to an application which will deserialize it, causing an invocation of readObject and trigger the attacker's payload. Since the attacker has complete control on the deserialized data, he can choose among all the Java classes present in the target application classpath, and manually compose them by using different techniques (e.g., wrapping instances in serialized fields, using reflection), and create an execution path that forces the deserialization process towards a specific target (e.g., execution of a dangerous method with input chosen by the attacker).

Recently researchers have published a paper that creates an automatic tool for generating a deserialization attack exploit for .NET applications [21]. Such an approach applies a practical field-sensitive taint-based dataflow analysis targeting the CIL languages. The core of such analysis leverages inter-procedural abstract interpretation based on method summaries, pointer aliasing, and efficient on-the-fly reconstruction of the control flow graph. This method is very specific for the CIL bytecode and it has not been tested on programming languages that use a different low-level representation such as Java bytecode.

Despite clear differences between Java bytecode and CIL, such representations also have similarities: both are low-level, object-oriented languages and they store objects on the heap. Even though it is tempting to create an equivalence between the two representations (e.g, applying the same analysis approach on both low-level languages), such goal is not easy to achieve since the two low-level representations work on languages with different characteristics (e.g., memory operations, safe pointer etc.). Such translation introduces issues about the exact meaning of equivalence between CIL and its translation into Java bytecode. Moreover, the translation should not introduce code artifacts that confuse the analyzer and consequently produce false positives and/or false negatives [6].

In this paper we face the problem of detection of the deserialization vulnerabilities for the Java language. Our purpose is to devise an automatic methodology that works directly on the language features and is able to output a potential deserialization attack chains. Our methodology uses a different approach compared with the one designed in [21]. In particular the analysis framework works directly on constructs of the Java language: objects/method, data type etc. and it designs an analysis which aims to discover potential attack Java deserialization chains that connect Java classes libraries. The advantage of working on the language construct is the use of semantic information that can be extracted by the rules of the language itself. As we will see in Sect. 3 such information is used to improve the precision of the analysis and reduce false positives and false negatives.

To this end we design a custom static data-flow analysis framework called JChainz, that works directly on the Java language and combines the reaching definitions and type propagation analysis for obtaining potential deserialization attack vectors, such attack vectors they need to manually validate for obtaining the final exploit. In particular our tool works in two main phases. In the first phase the system builds up a call graph and data dependency graph that contain the control-flow and data dependency information among the Java variables of

the analyzed code (e.g., libraries of the target application). Goal of this phase is to select execution links between the methods of different classes (e.g. execution chains) in terms of execution call. Then, in the second phase, the system analyzes the potential attack chains and validates them. The validation is applied by propagating the variable type in the graph and marking the type inconsistencies. Such techniques can help the system to exclude the majority of false positives and select potential real attack vectors.

We test our techniques against the most common Java libraries used in web technology. More precisely we select libraries from Apache Commons Collections version 3.1 and 4.0. The Commons Collections libraries are included in a great number of projects like for example on Apache Maven Central, where we can find more than 2700 public artifacts that use such libraries. The results of our experiments show our tool was able to validate the attack chains for the majority of already known vulnerabilities, and it was also able to discover multiple novel chains [16,17], new attacks that it has been confirmed in April 2022 by yososerial research community [9].

In summary, the paper reports the following contributions:

- We present a systematic approach for discovering deserialization vulnerabilities in Java applications including the framework and libraries that is based on custom program analysis techniques.
- We design and develop a tool that is able to extract a deserialization attack vector from the Java code and help the security analyst to fix the code of the vulnerable applications. Our tool is open sourced (https://github.com/Kigorky/JChainz) for future research.
- Our experiments show the effectiveness of our approach on finding new attack vectors. We describe three new case study attacks and will also discuss the limitations of our analysis and future improvements that consider reflection and dynamic proxying techniques.

2 Background

In this section we describe background concepts for understanding the security problems of deserialization of untrusted data in Java and we provide a real attack example.

2.1 Deserialization Terminology

Java Object Serialization. Serialization is the process of encoding objects into a stream of bytes, while deserialization is the opposite operation. Java deserialization is performed by the class `Java.io.ObjectInputStream`, and in particular by its method `readObject`. A class is suitable for serialization/deserialization if the following requirements are satisfied [19]: (1) the class implements the interface `Java.io.Serializable`, (2) the class has access to the no-argument constructor of its first non-serializable superclass. A class `C` can specify custom behavior

for deserialization by defining a `private void readObject` method. If present, such a method is called when an object of type `C` is deserialized. Other methods can be defined to control deserialization process: (1) `writeObject` is used to specify what information is written to the output stream when an object is serialized (2) `writeReplace` allows a class to nominate a replacement object to be written to the stream (3) `readResolve` allows a class to designate a replacement for the object just read from the stream.

2.2 Running Attack Example

To describe an example of a real attack, we use a real vulnerability present in Apache Common Collection libraries, and we show how an attacker can pilot a deserialization process and execute a dangerous native method. In Listing 1.2 we report the code for functions `heapify`, `siftDown` and `siftDownUsingComparator` of class `Java.util.PriorityQueue` of Java Framework. In Listings 1.3 and 1.4 we show methods `compare` of class `TransformingComparator` and method `transform` of `InvokerTransformer`, from library Apache Commons Collections 4. Listing 1.5 shows an hypothetical target class for executing a system command.

Listing 1.1. readObject in Java.util.PriorityQueue

```
private void readObject(Java.io.ObjectInputStream s) /* function a */
    throws Java.io.IOException,
           ClassNotFoundException {
  // Read in size, and any hidden stuff
  s.defaultReadObject();

  // Read in (and discard) array length
  s.readInt();
  queue = new Object[size];

  // Read in all elements.
  for (int i = 0; i < size; i++)
    queue[i] = s.readObject();

  heapify();
}
```

Listing 1.2. heapify and siftDownUsingComparator in PriorityQueue

```
private void heapify() { /* function b */
  for (int i = (size >>> 1) - 1; i >= 0; i--)
    siftDown(i, (E) queue[i]);
}

private void siftDown(int k, E x) {
  if (comparator != null)
    siftDownUsingComparator(k, x);
  else
    siftDownComparable(k, x);
}

private void siftDownUsingComparator(int k, E x) {
  int half = size >>> 1;
  while (k < half) {
    int child = (k << 1) + 1;
    Object c = queue[child];
    int right = child + 1;
    if (right < size &&& comparator.compare((E) c, (E) queue[right]) > 0)
    c = queue[child = right];
    if (comparator.compare(x, (E) c) <= 0)
    break;
    queue[k] = c;
    k = child;
  }
  queue[k] = x;
}
```

Listing 1.3. TransformingComparator.compare

```
public int compare(final I obj1, final I obj2) { final O value1 =
  this.transformer.transform(obj1); final O value2 =
  this.transformer.transform(obj2); return
  this.decorated.compare(value1, value2); }
```

Listing 1.4. InvokerTransformer.transform

```
public O transform(final Object input) {
  if (input == null) return null;
  try {
    final Class<?> cls = input.getClass();
    final Method method = cls.getMethod(iMethodName, iParamTypes);
    return (O) method.invoke(input, iArgs);
  ...
}
```

Listing 1.5. Command class

```
public class Command implements Serializable {
  private String command;

  public Command(String command) {
    this.command = command;
  }

  public void execute() throws IOException {
    Runtime.getRuntime().exec(command);
  }
}
```

Listing 1.6. Attack payload

```
final InvokerTransformer transformer =
  new InvokerTransformer("execute", new Class[0], new Object[0]);

final PriorityQueue<Object> queue =
  new PriorityQueue<Object>(2, new TransformingComparator(transformer));

queue.add(1);
queue.add(new Command("rm -f importantFile"));
```

Now, suppose an attacker created and serialized an object in listing 1.6. When this object is deserialized, the first method invoked after reading all the data from the priority queue is `heapify` as defined in the source code Listing 1.1 (readObject entry point of the deserialization); then `siftDownUsingComparator` is called (via `siftDown`), Listing 1.2, which uses the comparator modified by the attacker into the serialized object, in this case a `TransformingComparator`, Listing 1.6, for comparing the queue elements. The `compare` function in `TransformingComparator`, Listing 1.3, uses the field `transformer`, provided by the attacker, Listing 1.6, and calls its `transform` function on the objects being compared. At this point the `InvokerTransformer` is invoked, Listing 1.4, and such a method uses reflection to call the method with name equal to its field `iMethodName` on `input`, in this case the `Command` method. The reflection helps the attacker to invoke methods of generic classes; by crafting the deserialization input, the attacker is able to invoke method `execute` on an instance of the `Command` class with controlled parameters and execute arbitrary commands. In Listing 1.7 we report the stack trace collected at the execution of `Runtime.exec`, which contains all the Java methods invoked during the malicious deserialization event.

Listing 1.7. Stack trace of sample attack payload

```
Runtime.exec
Command.execute
Method.invoke
InvokerTransformer.transform
TransformingComparator.compare
PriorityQueue.siftDownUsingComparator
PriorityQueue.heapify
PriorityQueue.readObject
```

The attack vector described in this section is based on payload CommonsCollections2 from the ysoserial repository, used in real attacks. The only difference with the original version is the class `Command`, that we introduced for simplicity in our description. The real attack vector uses a dynamic class loading [9] as a gadget attack execution.

3 Overview

The goal of our analysis is to discover, given a specific Java library, the relationship among its classes and their methods in terms of execution. To discover such chains, as a first step we need to build a call graph that shows the relationships between methods of the analyzed classes in terms of caller and callee. Afterwards, we need to extract from the call graph, chains that reach an exit point of our interest (e.g., invoked method) and represent a potential attack vector. In Fig. 1 we depict an architectural design of our framework. More precisely, in our framework, we identify two main components: The **Finder** and the **Analyzer**. The **Finder** component starts from the Java bytecode and builds up the call graph (Sect. 3.1) of the target libraries by using the entry and exit point of any potential attack vectors (i.e., first three blocks in the diagram). Entry and exit points are defined by the deserialization process and the target attack class (Sect. 2.1). When the step is completed, the **Analyzer** component applies, for each discovered chain, the Data Dependencies Graph (Sect. 3.3) to determine the input data flow among the chain classes. In the last step of the analysis, the **Analyzer** applies a type propagation algorithm (Sect. 3.3) to exclude false positives and select the attack vectors candidates.

3.1 Call Graph Accuracy

The first challenge to solve is related to the call graph generation. A trivial solution for such a problem is to check the invoke instructions in Java bytecode, and build the call graph from them. While this represents a good starting point from our analysis, it is not sufficient to construct precise relationships among methods.

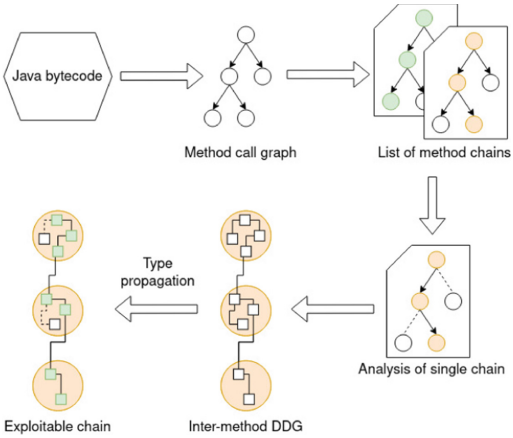

Fig. 1. Architecture of JChainz Framework

Listing 1.8. SubClass Example

```
class Example {
    SomeClass o = new SubClass();
    public a() {
        o.method();
    }
}
```

For example, consider the code in Listing 1.8. The call on o.method is performed on an instance of `SomeClass`, so the link `Example -> SomeClass` is trivial. At runtime, the instance is actually of type `SubClass`, so this link must also be considered. *Such missing information (i.e., runtime subtypes of classes and interfaces) can lead to an incomplete graph (e.g., missing chains' links), and produce false negatives since such class type is not considered and the attack vector cannot exploit its methods for executing the exploitable chain.* For this reason we need to consider such cases for building the graph, and include interface implementors and class extenders as well. More precisely, we create a link between methods in the graph only when at least one of the following conditions are satisfied:

- (1) The method's class implements the `Serializable` interface.
- (2) The callee method's class is a superclass of the caller method class.
- (3) The method has the `static` modifier.

It is important to note that all the objects (i.e., methods) that appear in the chain should be serializable. The only exceptions to such a case are invocations to methods in a non-serializable superclass (condition 2), or calls to static methods by directly invoking the method from the Java class (condition 3). For building the class call graph we use Soot [22] and we leverage Soot's capability (i.e., Soot APIs) of constructing the call graph of our input class path. In particular Soot first generates the Intermediate Representation (IR) for all the classes and their methods, and then it builds the call graph from Java invoke statements. For

any invoke statement we considered class extenders and interface implementors for the analyzed callees and we label the graph according to the discovered information.

3.2 Data Type Inconsistency

Once we have built the call graph, the system extracts the execution chains and validates them. The validation process defines the input data flow from the entry point (e.g., Serializable class) of the chain till the exit point (e.g., invoke method). More precisely, the Analyzer needs to control the existence of a data flow path that depends on the input and can be used for controlling the execution of the target attack class. Such analysis needs to exclude false positives that can be raised by data type inconsistency.

Listing 1.9. Data Type Inconsistency

```
1  class Example {
2      public example() {
3          return "FOO" + "BAR";
4      }
5  }
6
7  // class StringBuilder
8  public StringBuilder append(Object var1) {
9      return this.append(String.valueOf(var1));
10 }
11
12 // class String
13 public static String valueOf(Object var0) {
14     return var0 == null ? "null" : var0.toString();
15 }
```

To see an example of data type inconsistency problem, we consider the code in Listing 1.9. In this case, we have method `Example.example`, which concatenates two strings by using the method `StringBuilder.append`, and then we have the second method `String.valueOf` that returns the string value. A correct call graph must link them, and the following chain results in a correct execution stream, as a call to `Example.example` always results in the execution of the entire chain:

```
C1: Example.example -> StringBuilder.append -> String.valueOf
```

Analyzing the call graph we see that `valueOf` calls the method `toString` on its `Object` parameter. In such a case to obtain a valid attack vector, we should (e.g., attacker point of view) be able to assign an instance of `Object` (i.e. anything we want) to the parameter, and proceed from there. While this reasoning would be correct if we were considering only the method `String.valueOf`, in our case such an example produce a false positive.

```
C2: Example.example -> StringBuilder.append -> String.valueOf ->
Example.toString
```

In fact such a chain misses two important pieces of information for being correctly validated: (1) the type of the parameter `var0` can only be set to `String` (propagated from `Example.example`) and not to a general object, (2) moreover the string parameter is constant as defined in the class Example.example, `"BAR"`, and consequently it cannot be modified by the attacker.

3.3 Validation Algorithm

To solve the data type inconsistency problem and validate the attack chain we need to design a custom static data-flow analysis algorithm, that combines reaching definitions and type propagation analysis and works directly on the data. The idea is to build a data dependency graph (DDG), that contains information on control flow and data dependency between variables. By propagating the variable types in the graph, we can mark type inconsistencies and remove the false positives.

Data Dependency Graph. For implementing an accurate data type propagation analysis we need to apply intra and inter method mechanisms. In the following we report how the standard algorithm works in our specific context. We first apply the intra method mechanism by considering the following steps:

- For each method in the chain, we generate the control-flow graph (CFG), and trace the data dependency starting from a reaching definitions analysis [15] (**intra-method analysis**)
- For each link in the chain `M1 -> M2`, corresponding to a call to `M2` in the body of `M1`, we map the arguments in the call statement in `M1` to the corresponding variables in `M2` (**inter-method analysis**)

Each node in our DDG represents a particular variable defined in a specific statement. More in detail we define a node composed by a triple $(Method, Value, Unit)$, with the following parameters: (1) $Method$: represents the class and method of the current statement. (2) $Value$: represents the variable of the node. (3) $Unit$ represents the current statement. For example, considering the `return` statement at line 9, Listing 1.9, we can see that such statements affect two variables: `this` and `var0`. Therefore, two nodes in the DDG will be created.

We now define edges in the DDG, which represent dependency relationships between nodes. Such definitions are useful for constructing the intra-method representations. More precisely, there is an edge between node A and node B if A depends on B, the dependency is defined by the following rules:

1. A use of a variable V at a node N (with value V) depends on the definition of V at the node M
2. The definition of a variable V at a node N depends on the use of another variable U at a node M if N and M have the same unit
3. A use of a variable V at a node N (with value different U different from V) depends on the definition of U at the node M

Listing 1.10. DDG construction example

```
class Example {
    public void a() {
        String var0 = "BAR";
        String var1 = "FOO" + var0;
    }

    public void b() {
        new Example().c("FOOBAR");
    }

    public void c (String var2) {
        String toPrint = var2;
        this.a();
        System.out.println(toPrint);
    }
}
```

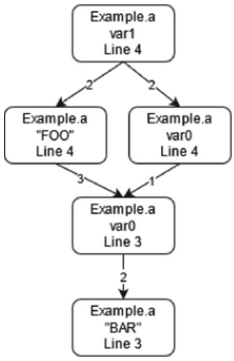

Fig. 2. intra-method DDG for Listing 1.10

In Fig. 2 we show the intra-method DDG constructed for method `Example.a` in Listing 1.10. Each edge is marked with the rule applied for data dependency. At this point, the DDG contains only information about intra-method data dependency;

After we build a intra-method we have to insert inter-method information to the graph.

To this end we follow the following strategy: when we find an invoke statement at method M_x in the chain, we check whether the callee belong to the step M_{x+1} of the chain. If this is the case, we create an inter-method edge in our DDG. In particular in our context we need to distinguish two main cases:

1. inter-method parameter call - in this case, the value of node M_x is a parameter of the method call. We track the value and make sure it is correlated with the appropriate parameter in the next method's DDG.
2. inter-method instance call - in this case, the value of node M_x is the object on which the method call is performed. Therefore, in the CFG of M_{x+1}, such object will be referenced by the `this` pointer in Java.

Type Propagation Analysis. After computing the DDG, we execute a type propagation analysis. To this end we first assign type information to each node for which the type is known, and we then propagate the information through the DDG, to detect any type inconsistencies. For this purpose, we add a dictionary to each node, named *allowed_types*. This dictionary contains an entry for each known variable at a given node in the DDG, and contains all the possible types for this variable; the types are inferred from the DDG itself.

We start with a `null` value for *allowed_types* at every node, then we initialize only nodes with no dependencies for their value, apply the following rule: for each node N with value V and no dependencies for V, we add the type of V to the *allowed_types* for V at N.

Once we statically determine the type information for each node, we navigate the graph and for each step we process all the nodes which have no dependencies with *allowed_types*. When we process node N with value V, we copy the allowed types for each variable in its successors in its *allowed_types* dictionary (duplicates are removed), with the following logic: each of them is compared with the type T of V at N; only types that "can hold" T are copied and allowed for V at N (i.e., T and supertypes). If a node with value V has the empty set as the allowed types for V, we have found a type inconsistency, consequently the data-flow through a particular node is not possible.

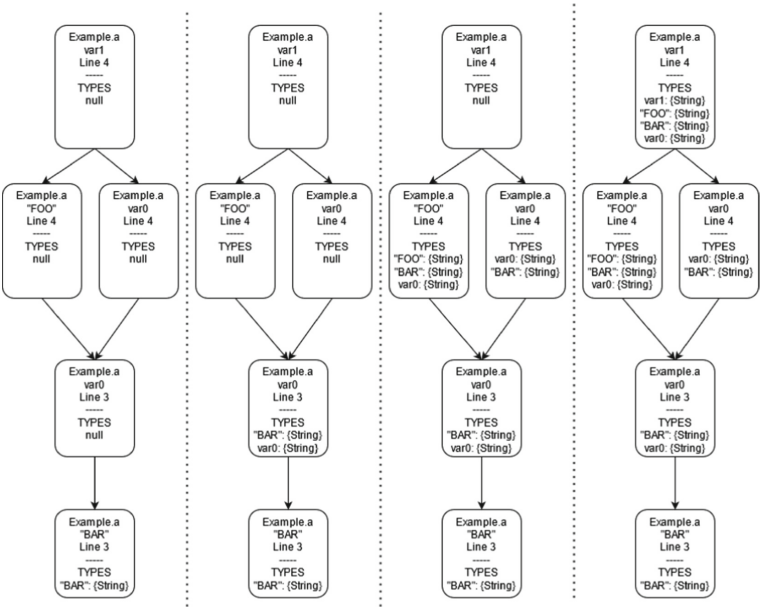

Fig. 3. Type propagation algorithm for method `a` in Listing 1.10. Subsequent iterations are shown from left to right

Special care is taken for inter-method links, which are handled separately. The logic of type propagation is the same, but the types are matched also on the called object and the method parameters, depending on the type of inter-method edge, described above. For instance, in the inter-method DDG shown in Fig. 3, our algorithm correctly infers type `String` for `var2`, and type `Example` for `this`. The algorithm iterates till all the nodes have been processed. Then all the nodes that are marked type inconsistency are removed from the graph. The remaining chains are marked as attack vector candidates.

4 Experimental Evaluation

In this section we present our evaluation results. We evaluate both the components of our system, the Finder and the Analyzer, measuring their effectiveness and performance. For validating the results of JChainz we also perform data analysis for discovering false positives and false negatives along with the real attack vectors. In Appendix section we also reported two case studies of new exploitation chains found by our framework. The following experiments have been run on a Debian GNU/Linux 9.11 (64 bit) machine with an Intel Xeon CPU (2.27 GHz) and 20 GB of DRAM. The code was compiled using Java OpenJDK 1.8.

4.1 Dataset

To evaluate our framework, we select the test code from the Commons Collections libraries 3.1 and 4.0 as reported in the ysoserial repository [9]. The Commons Collections packages used for the evaluation are composed of: (1) 421 classes and 3485 methods in the Commons Collections 4.0; (2) 425 classes and 3728 methods in the Commons Collections 3.1. In literature, those libraries are known to be vulnerable, thus representing an interesting target for the evaluation of our tool (e.g., Ground Truth). Due to their versatility, the Commons Collections libraries are included in a huge number of projects, for example in Apache Maven Central project. Moreover Commons Collections libraries seek to build upon the JDK classes by providing new interfaces, implementations and utilities also in a web context [7].

4.2 Finder Results

In this section we report the results of a run of the Finder on our dataset. Main aim of the Finder is to discover connections among the methods defined in a specific library. We set up the tool to perform a depth first search navigation starting from the `readObject()` custom implementations (i.e. entry points), reaching `Java.lang.reflect.Method.invoke` (i.e. exit point), which is an easy springboard for an attacker to launch arbitrary code. To help our Finder component on performing its own task we set up several parameters that control the graph's analysis exploration. In particular we define the following variables: (1) *Max depth*: maximum depth for the DFS algorithm for exploring the call graph

in terms of nodes starting from a single entry point.(2) *Max chains*: maximum numbers of chains devised from a single entry point. (3) *Max seconds*: Maximum number of seconds to search for chains starting from a single entry point.

Table 1. Finder parameters

Parameter	Value
Max depth	10
Max chains	100
Max seconds	10800

In Table 1 we reported the values considered in our experiments. *The* `max depth` *parameter has been selected by considering the round up average depth of the already known real exploitable chains from entry points to the exit points (e.g, Common Collections in ysoserial repository).* Note that, the arguments `max_chains` and `max_seconds` are mutually exclusive parameters that interrupt the research before the full graph exploration is completed. We chose such values based on the ground truth chains parameters. In our experiments, the searching phase took approximately $40h$ to complete. In Table 2 we report the results of the Finder's Analysis on a single run on both libraries.

Table 2. Finder results

	CC 3.1	CC 4.0
Entry points	32	30
Active Entry Points	11	12
Total number of chains found	36	567

In particular, in the Table 2 we reported three main parameters of our results: (1) number of *Entry points* that have been statically found in the libraries. With the term "entry point", we considered the `Serializable` classes that redefine the `readObject()` method. (2) *Active Entry points*: this is a subset of entry points that contains at least one chain found by the Finder component. It is important to note that due to the time constraints of the graph exploration not all the Entry points have been analyzed in a single run of the experiment. (3) *Total number of chains found*: the number of chains for which the Finder generates a path from the entry point to an exit point (i.e., Method.invoke).

In summary, in this first run of the experiments based on the previous settings parameters, the Finder analyzed a class's call graph composed by 934300 connections (i.e., arches) for common collection 3.1 and a class's call graph composed by 519980 connections for common collection 4.0 and it was able to extract 36 chains for common collections 3.1 and 567 chains for common collections 4.0.

4.3 Analyzer Results

After the searching phase performed by the Finder component, the system sends all the found chains to the second component, the Analyzer, whose main aim is to validate them. The Analyzer starts analyzing the single chain and for each of them builds up the inter-method DDG, then it applies the data types propagation algorithm described in the previous section. *The whole analysis took approximately 27h.* In the Table 3 we report the results of these second running steps.

Table 3. Analyzer results

	CC 3.1	**CC 4.0**
Non-exploitable chains	22	531
Exploitable chains	8	36

As we can see from the Table 3, the Analyzer was able to exclude a large fraction of the false positive chains. In particular for the CC 4.0 the tool was able to discard 93% of the false positive chains while for the 3.1 the tool was able to discard 64% of the false positive chains. *This reduction was mainly achieved by the data type propagation analysis. In particular the average size of the Data Dependencies Graph before the pruning for common collection 3.1 was composed by 658 nodes and 1338 arches. After the pruning we obtained a graph with on average 76 nodes and 143 arches. For the common collection 4.0 we start with a graph before pruning composed by 718 nodes and 1468 arches and we obtain a graph with 138 nodes and 249 arches.*

To confirm the results we apply the manual analysis on the 44 exploitable chains found by our tool. The manual analysis reveal that 32 of the 44 exploitable chains represent false positive and 12 of the 44 chains were real attack vectors. Among the results of the exploitable chains, we first search for the ground truth exploits present in Common Collections Libraries 3.1 and 4.0. The Analyzer component was able to validate all of them with three missing exploits. Such an issue about the results depends on the fact that our tool does not handle Java Proxy classes which alter the program behavior at run-time (i.e., dynamic feature). Such dynamic gadgets are needed for exploit CC1, CC3, CC4. This represents a limitation for our tool and it will be discussed in the limitations section.

Beyond the already known exploits our tool was able to find three new real attack chains, the exploit CC7, CC8 and CC10 that have been acknowledge by the yoserial community [9] and *the other six new exploit discovered by our tool are variants of the original ones. For space limitation we report some exploits of the variants here:* https://github.com/Kigorky/JChainz/tree/main/exploits. CC7, CC8, CC10 can be considered new since they require a new exploitation technique for delivering the attack. In Appendix section we report a description on how to build up a successful attacks by using such chains.

In Table 4 we report all the known vulnerabilities validated by our tool.

Table 4. Ground Truth ChainzAnalyzer results

Vulnerability Exploit	Results
CC1 exploit	Failed
CC2 exploit	Pass
CC3 exploit	Failed
CC4 exploit	Failed
CC5 exploit	Pass
CC6 exploit	Pass

The 32 false positive chains were present in the results show the problem related to the precision of the analysis of our framework. Through the manual analysis we find out the typical false positive that our tool is affected. In particular our Analyzer is not able to process the expression of the conditions' in terms of value. In the following code we report a case of false positive found in our experiments. In particular this function is a part of the exploitable chain validated by the Analyzer.

Listing 1.11. False Positive example

```
1
2   private GeneralRange(Comparator<? super T> comparator, boolean hasLowerBound, @Nullable T
        lowerEndpoint, BoundType lowerBoundType, boolean hasUpperBound, @Nullable T upperEndpoint,
        BoundType upperBoundType) {
3       ...
4       if (hasLowerBound) {
5           comparator.compare(lowerEndpoint, lowerEndpoint);
6       }
7       ...
8   }
```

In our exploitable chain the system includes the method compare of the `Comparator` class defined as a block of the if statement (Line 5). The problem here is that the `hasLowerBound` is always set to false by the class defined in this chain and the method `comparator.compare()` will never be executed. Consequently since our chain cannot reach that method, the exploit is not feasible.

5 Limitations

Our tool is affected by some limitations, mainly due to the technical limits of the static analysis approach. The dynamic features of Java language, such as the reflection technique, are known to be an obstacle to the static code analysis. Due to the nature of these objects, the tool is not able to detect chains that could potentially be exploitable (e.g., false negatives). For example, our tool cannot handle proxy classes which alter their behavior at run-time. Some tools tried to model the static analysis over these dynamic features but this problem is still quite hard to solve [8,13].

Moreover our tool cannot handle the expression evaluation of the conditions statement (i.e., false positive). In particular such a problem could be solved by

adding more precising analysis like symbolic execution. At the moment, several possibilities exist for performing symbolic execution in Java [1,2]; however, while constraint solving works well with basic types such as integers and strings, to the best of our knowledge there is currently no modeling of custom objects in OOP. If such a model were developed, then the whole search of exploitable chains could be made more accurate, by exactly solving constraints on objects and variables, and deterministically generating inputs that allow a particular chain to be executed/exploited. Another point for improving our analysis is to use a new framework for building up Java Call Graph such as [20].

6 Related Works

The most recent work related to ours is by Shcherbakov and Ballium. [21]. In their work, the authors present a tool, SerialDetector, aimed at automatic discovery of Object Injection Vulnerabilities in .NET applications and libraries. Such an approach is based on the CIL intermediate language and based its own efficacy on a practical field-sensitive taint-based dataflow analysis targeting the CLI languages. This method is very specific for the CIL bytecode and it has not been tested on programming languages that use a different low-level presentation such as Java (e.g., bytecode).

In the particular context of deserialization vulnerabilities attack, an interesting work to mention is the tool Serianalyzer by Bechler [18]. Serianalyzer uses static Java bytecode analysis to trace native method calls made during the deserialization process and it uses several heuristics to identify already known attack patterns. Although it produces many false positives, it has been used to find many of the exploits present in the ysoserial repository. In our work we decided to create a more agnostic tool that leverages the capabilities of Soot and its intermediate representation. In particular we design an automatic analysis by implementing ad-hoc data flow and type propagations analysis to discover such a tool.

On the protection side several attempts have been made for protection against attacks based on deserialization of untrusted data. Dietrich et al. [5] analyze the problem of deserialization of untrusted data not only in Java, but in several affected languages. After analyzing a few chains that cause Denial Of Service, they study in detail possible mitigation for the problem. In the specific context of Java deserialization, Cristalli et al. [3] describe a system for establishing the trusted execution path in an existing application during a learning phase, and enforcing it at run-time with analysis of stack traces in the JVM. A similar approach had been followed by Hawkins et al. [10]; their ZenIDS system uses trusted execution path validation for protection of PHP software via anomaly detection. ObjectMap [12] is a tool that aims at detecting vulnerable deserialization entry points in Java and PHP systems. Most of those works check dynamically the integrity features of the deserialization process and show a quite big run-time overhead. The final goal of our tool is to recognize and directly correct the vulnerabilities inside the Java source code.

7 Conclusion

In this paper we present a new tool, called JChainz, that is the first tool that directly work on the Java language and it is able to discover untrusted data deserialization attack vector. We present the first systematic approach for automatic creating the Deserialization attack in Java applications including the framework and libraries. We test our tool on well-known libraries and we show its effectiveness by validating results on known and new vulnerabilities. We describe three new case study attacks along with the limitations of our approach and future improvements such reflection and dynamic proxing.

Acknowledgment. This project has received funding by the Italian Ministry of Foreign Affairs and International Cooperation (grant number: PGR00814).

1 Appendix

1.1 Case Studies

By taking advantage of our tool, we discovered and exploited new chains described in the following repositories [16,17]. Each chain is composed of two main parts, the first one from the entry point to the exit-point. In this case, the exit-point is the `method.invoke` method. The latter exit-point allows an attacker to access and call the entire set of methods and classes available in the java classpath. The second part of the chain is composed of a gadget. In our experiments, we attached the well-known gadgets already available in the ysoserial repository, which allowed us to run arbitrary code. The gadget can be seen as an already sequence of methods for achieving a specific operation. These chains have been discovered by the Finder, filtered by the Analyzer, then manually validated and exploited.

CommonsCollections7 The payload CommonsCollections7 [17], found with the aid of our tools, consists of the following chain:

```
java.util.Hashtable.readObject
java.util.Hashtable.reconstitutionPut
collections.map.AbstractMapDecorator.equals
java.util.AbstractMap.equals
collections.map.LazyMap.get
collections.functors.ChainedTransformer.transform
collections.functors.InvokerTransformer.transform
java.lang.reflect.Method.invoke
sun.reflect.DelegatingMethodAccessorImpl.invoke
sun.reflect.NativeMethodAccessorImpl.invoke
sun.reflect.NativeMethodAccessorImpl.invoke0
java.lang.Runtime.exec
```

The chain starts in the JDK class `Hashtable`, and produces an invocation of an arbitrary system command, via `Runtime.exec`. In order to reach this result, the chain reuses the `LazyMap` gadget from chain CommonsCollections5, already part of ysoserial before our work. Therefore, the novelty of CommonsCollections7 consists of the *trigger* made of the first five methods in the chain, up to the invocation of the *gadget* with entry point `LazyMap.get`.

While the potential exploitability of the chain was confirmed by our Analyzer, we still had to build the code for the exploit. To trigger the method sequence leading to the invocation of `LazyMap.get` starting from `Hashtable`, we built an hashtable containing two instances of the `LazyMap` gadget object we wanted to reuse, with the aim of triggering comparison between the two in the hashtable upon the insertion of the second. This comparison would force the call to `equals` on the `LazyMap`, which calls method `get` and triggers the gadget.

We discovered that inserting the same object twice in the hashtable was not sufficient, as the duplicate would be recognized right away without the need of any comparison with the objects already present in the hashtable. Therefore, we fabricated two *different* instances of the `LazyMap`, but with *colliding hashes*. This was possible because it is extremely easy to obtain colliding object hashes in Java, as the hashing mechanism has not been designed for security purposes and does not make use of any cryptographic hash function. In the specific case of `LazyMap`, the hash of the entire object is calculated from the hashes of the objects in the map. Therefore, it was sufficient for us to make the keys of the `LazyMap` gadgets collide. In particular, we chose colliding String objects `"yy"` and `"zZ"`.

At this point, the `LazyMap` objects can be inserted in the `Hashtable`, which will be then serialized. When deserialized, the reconstruction of the hashtable via its custom `readObject` method will insert the two objects. The insertion of the second will trigger a comparison with the first because of the colliding hashes, starting the rest of the chain as seen above. This manual design enabled us to transform the chain found by our framework into a fully functional deserialization exploit.

CommonsCollections8. The payload CommonsCollections8 has an interesting property that differentiates it from all other previous Commons Collections payloads: its entry point (i.e. the serializable class `TreeBag`) is part of the library itself, while all other known chains have entry points in standard Java classes found in the JRE. The payload CommonsCollections8 [16] generates the following stacktrace:

```
  org.apache.commons.collections4.bag.TreeBag.readObject
collections4.bag.AbstractMapBag.doReadObject
java.util.TreeMap.put
java.util.TreeMap.compare
collections4.comparators.TransformingComparator.compare
collections4.functors.InvokerTransformer.transform
java.lang.reflect.Method.invoke
```

```
sun.reflect.DelegatingMethodAccessorImpl.invoke
sun.reflect.NativeMethodAccessorImpl.invoke
sun.reflect.NativeMethodAccessorImpl.invoke0
com.sun.org.apache.xalan.(...).TemplatesImpl.newTransformer
... (TemplatesImpl gadget)
java.lang.Runtime.exec
```

This chain starts in the `TreeBag` class and leads to the execution of the `Runtime.exec` method, triggering the vulnerability in the Commons Collections 4.0 package. The contribution of this chain, like the previous one (Sect. 1.1), consists of spotting a new entry point.

The payload is composed by a `TreeBag` object built with a comparator of the type `TransformingComparator` and populated with a `TemplatesImpl` object from the ysoserial repository. During the deserialization process, the `TreeBag` class builds a new `TreeMap` object containing the attacker's comparator and passes it to the `AbstractMapBag.doReadObject` method as a parameter. At this point, the `put` method is invoked on the map object received as parameter, triggering the `compare` method call on the unsafe comparator. Starting from the `transform` method, the following operations that lead to the execution of arbitrary code are managed by the gadget from ysoserial.

References

1. Java Pathfinder. https://github.com/javapathfinder
2. Java Symbolic Execution. https://docs.angr.io/advanced-topics/java_support (2019)
3. Cristalli, S., Vignati, E., Bruschi, D., Lanzi, A.: Trusted execution path for protecting java applications against deserialization of untrusted data. In: Bailey, M., Holz, T., Stamatogiannakis, M., Ioannidis, S. (eds.) RAID 2018. LNCS, vol. 11050, pp. 445–464. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-00470-5_21
4. Dahse, J., Krein, N., Holz, T.: Code reuse attacks in PHP: automated pop chain generation. In: Proceedings of the ACM Conference on Computer and Communications Security, vol. 11, pp. 42–53 (2014)
5. Dietrich, J., Jezek, K., Rasheed, S., Tahir, A., Potanin, A.: Evil pickles: dos attacks based on object-graph engineering. In: 31st European Conference on Object-Oriented Programming (ECOOP 2017). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2017)
6. Ferrara, P., Cortesi, A., Spoto, F.: From CIL to java bytecode: semantics-based translation for static analysis leveraging. Sci. Comput. Program. **191**, 102392 (2020)
7. The Apache Software Foundation. Java collections framework. https://commons.apache.org/proper/commons-collections/
8. Fourtounis, G., Kastrinis, G., Smaragdakis, Y.: Static analysis of java dynamic proxies. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, pp. 209–220, New York, NY, USA. Association for Computing Machinery (2018)
9. Frohoff, C.: ysoserial repository. https://github.com/frohoff/ysoserial (2015)

10. Hawkins, B., Demsky, B.: Zenids: introspective intrusion detection for PHP applications. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pp. 232–243. IEEE (2017)
11. Holzinger, P., Triller, S., Bartel, A., Bodden, E.: An in-depth study of more than ten years of java exploitation, pp. 779–790 (2016)
12. Koutroumpouchos, N., Lavdanis, G., Veroni, E., Ntantogian, C., Xenakis, C.: Objectmap: detecting insecure object deserialization. In: Proceedings of the 23rd Pan-Hellenic Conference on Informatics, pp. 67–72 (2019)
13. Landman, D., Serebrenik, A., Vinju, J.J.: Challenges for static analysis of java reflection - literature review and empirical study. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pp. 507–518 (2017)
14. Lekies, S., Kotowicz, K., Groß, S., Nava, E.V., Johns, M.: Breaking cross-site scripting mitigations via script gadgets, Code-reuse attacks for the web (2017)
15. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer Publishing Company, Incorporated, Cham (2010)
16. Authors names obfuscated. Commonscollections8 (2019). https://github.com/frohoff/ysoserial/pull/116
17. Authors names obfuscated. CommonsCollections7 (2019). https://github.com/frohoff/ysoserial/blob/master/src/main/java/ysoserial/payloads/CommonsCollections7.java
18. Bechler, M.: Serianalyzer (2017). https://github.com/mbechler/serianalyzer
19. Oracle Corporation. The serializable interface (2017). https://docs.oracle.com/javase/8/docs/platform/serialization/spec/serial-arch.html#a4539
20. Santos, J.C., Jones, R.A., Ashiogwu, C., Mirakhorli, M.: Serialization-aware call graph construction. In: Proceedings of the 10th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis, SOAP 2021, pp. 37–42. Association for Computing Machinery, New York (2021)
21. Shcherbakov, M., Balliu, M.: Serialdetector: principled and practical exploration of object injection vulnerabilities for the web. In: Network and Distributed Systems Security (NDSS) Symposium 202121–24 February 2021 (2021)
22. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot - a java bytecode optimization framework. In: Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research, CASCON '99, p. 13. IBM Press (1999)