POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Evaluating the Testability of Insecure Deserialization Vulnerabilities via Static Analysis

Tesi di Laurea Magistrale in
Computer Science and Engineering - Ingegneria Informatica

Author: **Alessandro Sabatini**

Student ID: 953493
Advisor: Prof. Stefano Zanero
Co-advisors: Andrea Valenza
Academic Year: 2020-21

Abstract

The growing trend of web applications led to the increase of data exposed through the Internet.

In turn, this resulted a growing concern by the software industry to protect sensitive data contained in web applications. This sensitive data can be accessed from unauthorized subjects exploiting some flaws present in the program, that are called vulnerabilities.

In this thesis, we focus on one particular vulnerability: Insecure Deserialization. In particular, we apply static analysis to test the source code for the presence of Insecure Deserialization vulnerabilities.

The aim of this thesis is to create a benchmark application that evaluates the effectiveness of static security scanners when testing for Insecure Deserialization vulnerabilities.

In order to achieve this, we have made an experiment in which we have selected a sample of five static analysis tools and we have analyzed their behavior in relation to seven target web applications, vulnerable to Insecure Deserialization. Thanks to the results of this experiment, we have evaluated the performances of the tools and, learning from them, we have created our own web application vulnerable to Insecure Deserialization, called BenchStress, that deliberately blocks testing techniques. This will be a benchmark for testing the effectiveness of testing tools in detecting Insecure Deserialization Vulnerabilities.

Keywords: Insecure Deserialization, Deserialization of Untrusted Data, Benchmark, Static Analysis, BenchStress.

Abstract in lingua italiana

La rapida diffusione delle applicazioni web ha portato all'aumento dei dati esposti via internet.

A sua volta, questo ha provocato una crescente preoccupazione da parte dell'industria del software per proteggere i dati sensibili contenuti nelle applicazioni web. Questi dati sensibili possono essere accessibili da soggetti non autorizzati sfruttando alcuni difetti presenti nel programma, che sono chiamati vulnerabilità.

In questa tesi, ci concentriamo su una particolare vulnerabilità: la Deserializzazione Insicura. In particolare, applichiamo l'analisi statica per testare la presenza nel codice sorgente della vulnerabilità di deserializzazione insicura.

Lo scopo di questa tesi è quello di creare un'applicazione di benchmark che valuti l'efficacia degli scanner di sicurezza statica durante i test per le vulnerabilità di deserializzazione insicura.

Per raggiungere questo obiettivo, abbiamo fatto un esperimento in cui abbiamo selezionato un campione di cinque tool di analisi statica e ne abbiamo analizzato il loro comportamento in relazione a sette diverse applicazioni web target vulnerabili alla deserializzazione insicura. Grazie ai risultati di questo esperimento, abbiamo valutato le prestazioni dei tool e, imparando da essi, abbiamo creato la nostra applicazione web vulnerabile alla deserializzazione insicura, chiamata BenchStress, che blocca deliberatamente le tecniche di testing. Questa applicazione sarà un benchmark per testare l'efficacia degli strumenti di test nel rilevare vulnerabilità di deserializzazione insicura.

Parole chiave: Deserializzazione Insicura, Benchmark, Analisi Statica, BenchStress

Acknowledgements

Here you might want to acknowledge someone.

Contents

1 | Introduction

Web applications are the most popular way for delivering services via the Internet and have become essential in our daily lives. Due to their ever growing popularity and the high value data they expose, the software industry is increasingly paying attention to the aspects concerning their protection.
A vulnerability, in computer security, can be defined as a weakness in the computer system that leaves information exposed to unauthorized subjects. It can be described as a combination of three elements: the presence of a flaw in the system, the ability of the attacker to discover the flaw and the attacker's capability to exploit the flaw. The presence of a flaw could be due to several causes: the lack/misuse of best practices while coding (e.g., input/output validation); often the testing of security aspects is neglected in favor of functional requirements; given their performance overheads and possible false positives that may disrupt normal behavior, attack detection mechanisms are sometimes not included in the environment.

As is clearly visible from the Figure 1.1, the number of vulnerabilities during the years is continuously growing, in fact from the 2016 where the number of vulnerabilities overall was around 6608 in just five years it tripled and more reaching the number of 20137 vulnerabilities in the 2021 year.

While, regarding the 2022 year it is not comparable to the others because it consider just one month even if it is incredible to note that this number, in just one month, is already bigger than the 2004 year and all the previous years.

All the data used for Figure 1.1, Figure 2.1 and Table 2.1 have been retrieved from the National Vulnerability Database (NVD) [16]. The National Vulnerability Database (NVD) is the U.S. government repository of standards-based vulnerability management data represented using the Security Content Automation Protocol (SCAP). This data enables automation of vulnerability management, security measurement, and compliance. The NVD includes databases of security checklist references, security-related software flaws, misconfigurations, product names, and impact metrics.

In addition to providing a list of Common Vulnerabilities and Exposures (CVEs), the NVD

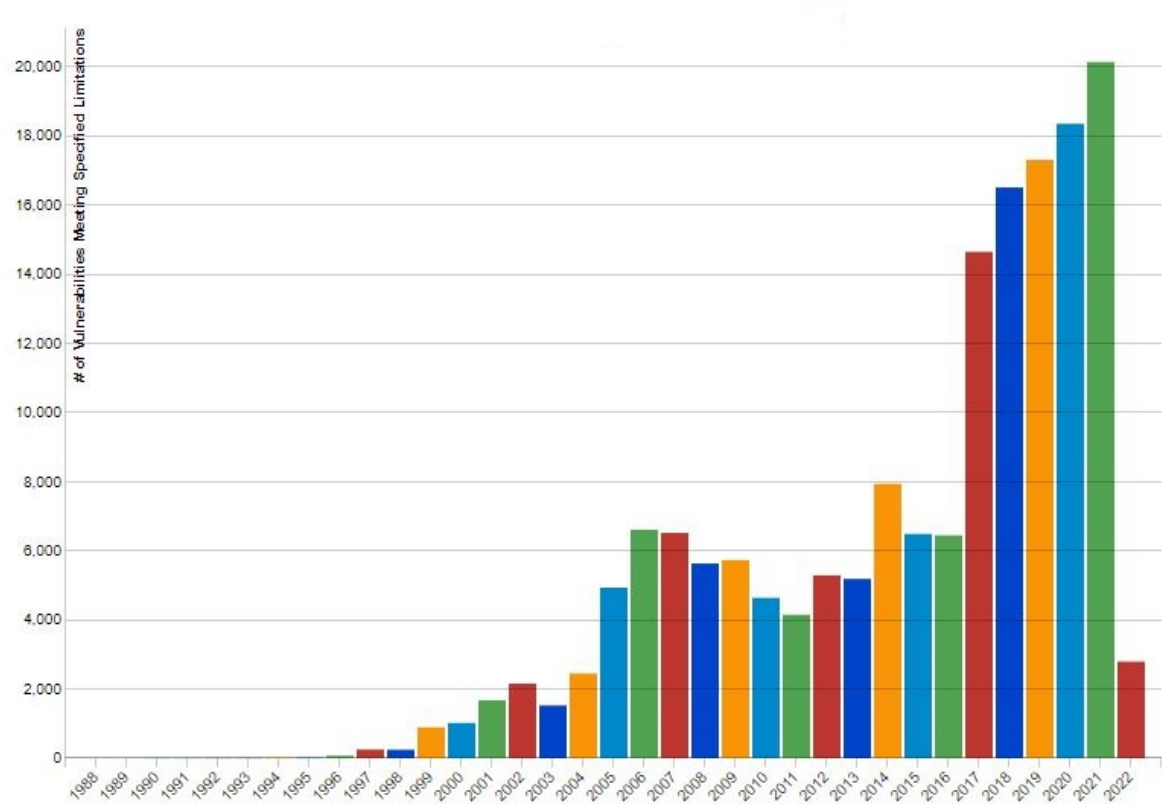

Figure 1.1: Trend of all vulnerabilities discovered during the years (data taken on February 2022).

scores vulnerabilities using the Common Vulnerability Scoring System (CVSS) which is based on a set of equations using metrics such as access complexity and availability of a remedy.

The Common Vulnerabilities and Exposures (CVE) program is a dictionary or glossary of vulnerabilities that have been identified for specific code bases, such as software applications or open libraries. This list allows interested parties to acquire the details of vulnerabilities by referring to a unique identifier known as the CVE ID. It has garnered increasing awareness in recent years, making it important for participants and users to understand the fundamental elements of the program.

Several security problems affect high-level software environments. These problems can emerge regardless of security measures applied to the underlying systems. This is the case, for example, of problems arising from deserialization of untrusted data in several programming languages, such as Java, PHP, Python, and C#, where a technology with many legitimate uses (data serialization) is exploited to obtain unintended and malicious behavior in software. By leveraging object deserialization, attackers are able to chain

pieces of benign software, leading to effects such as arbitrary code execution on the target system. Solving these problems while maintaining functionality is not trivial, because some of the causes that make the attack possible are needed key features of the serialization technology. Also, in the code chains mentioned above, it is hard to identify a single "culprit" that permits attacks; the malicious effects arise from the interaction of pieces of benign software, with no single point of failure.

The focus of this thesis is on one particular vulnerability: **Deserialization of Untrusted Data** or **Insecure Deserialization**.

Motivation

Deserialization of untrusted data is a cause of security problems in many programming languages. In Java, it might lead to remote code execution (RCE) or denial of service (DOS) attacks. Even though it is easy to check whether preconditions for this type of attack exist in an application (that is, deserialization performed on user-controlled data), designing and carrying out a real attack is a hard task, due to the complexity of creating the attack payload. In order to exploit this type of vulnerability, an attacker has to create a custom instance of a chosen *serializable* class which redefines the *readObject* method. The object is then serialized and sent to an application which will deserialize it, causing an invocation of *readObject* and triggering 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). There are several public exploits that show the impact of the attack on real Java frameworks, such as JBoss and Jenkins, which are based on several common Java libraries, such as Oracle JRE 1.7, Apache Commons Collection 3 and 4, Apache Commons BeanUtils, Spring Beans/Core 4.x and Groovy 2.3.x and more.

In this thesis we decided to make two important choices: (1) We decided to focus on only one particular programming language: **Java**. The motivation behind this choice is because historically Java has an ecosystem that exploits so much the deserialization and there are many libraries and applications already made which use it, while other languages use it less. (2) We applied the **static analysis** because this vulnerability is easier to find through the code, instead of a black box approach.

Goal

The goal of this project is to create a *de facto* benchmark application to evaluate the effectiveness of static security scanners, with a particular focus on Insecure Deserialization vulnerabilities. This benchmark application contains portions of code that static security scanners cannot easily navigate, thus possibly hiding the included Insecure Deserialization vulnerabilities.

To achieve this, an intermediate goal of this thesis is to understand how real-world automated scanners detect Insecure Deserialization vulnerabilities in Java source code. In fact, this thesis reports the results of the evaluation of 5 real-world static analysis tools in detecting Insecure Deserialization vulnerabilities in 7 vulnerable targets web applications, as well as our benchmark application.

Overview

The contents of the thesis are organized in five chapters, including the current introductory one.

In Chapter 2, we start with an overview on the background concepts needed to understand better this thesis. In particular, the Insecure Deserialization vulnerability is presented giving its definition, how to identify it and there is also an example of a vulnerable program with the related exploitation. After, some impacts and mitigations are presented in order to view how the libraries are affected by this vulnerability and the possible remediation to counter it. Finally, we give an explanation on the static analysis and the reason why we chose it.

Chapter 3 gives a description about the targets chosen as samples for the experiment and also the reason behind the selection of these targets. Furthermore, it is presented our own web application, called *BenchStress*, that will be used to test the tools showing some snippets of code and their description.

In Chapter 4, we have analyzed the results obtained from the experiment. At the beginning, we deal with the introduction of the tools used for this thesis. There is a list of the tools, and, then, they are presented one by one in detail paying attention to the patterns used by each tool to discover vulnerabilities that will be a section on its own called *Bug Patterns*. Next, it is explained how the experiment was conducted and under which conditions. Then, we show the results of the experiment in a table that is followed by some considerations and interpretations of the results. In particular, it is given a lot

of attention to the results when the tools were launched on our benchmark program.

Chapter 5 summarizes the most relevant results achieved in the thesis, pointing out the advantages and weaknesses associated to the found results. We also propose some extensions to the work.

2 | Background

In this chapter we describe background concepts for understanding the security problems with deserialization of untrusted data in Java. In particular, we briefly describe the definition of Insecure Deserialization vulnerability and the identification and attack surface estimation. Then, we present a vulnerability example and how the latter can be exploited to obtain malicious side effects when applied to untrusted data. Finally, some impacts and mitigations are described. After this, there is another section in which is explained the concept of static analysis and why it is important.

2.1. Insecure Deserialization vulnerability

In this section is presented the vulnerability and the reason why it happens.

This vulnerability is present from 2007, but it experienced a significant increase from 2017. In fact, in 2017 around 60 remote code execution (RCE) deserialization vulnerabilities were reported, not including deserialization issues that only impact the availability of a system (Denial-of-Service). In 2018, more than 100 such vulnerabilities have been reported (Table 2.1).

As is clearly visible from Figure 2.1, the Insecure Deserialization shows a growing trend in the last years reaching its maximum, until now, in 2021 where this particular vulnerability represented more than one percent of all the vulnerabilities present in that year.

The deserialization has gain a lot of attention in the last years because of its high impact and severity, in fact, it can have very dangerous effect. This is what made it earn a position in the OWASP Top 10 list of the 2017, which is the list of the ten most critical web application security risks [19]. The OWASP Top 10 is a book/referential document outlining the 10 most critical security concerns for web application security. The report is put together by a team of security experts from all over the world and the data comes from a number of organizations and is then analyzed. The Insecure Deserialization was added thanks to the community's effort that voted in a survey for industries and it was ranked at number three, so it was added to the Top 10 as A8:2017-Insecure Deserialization

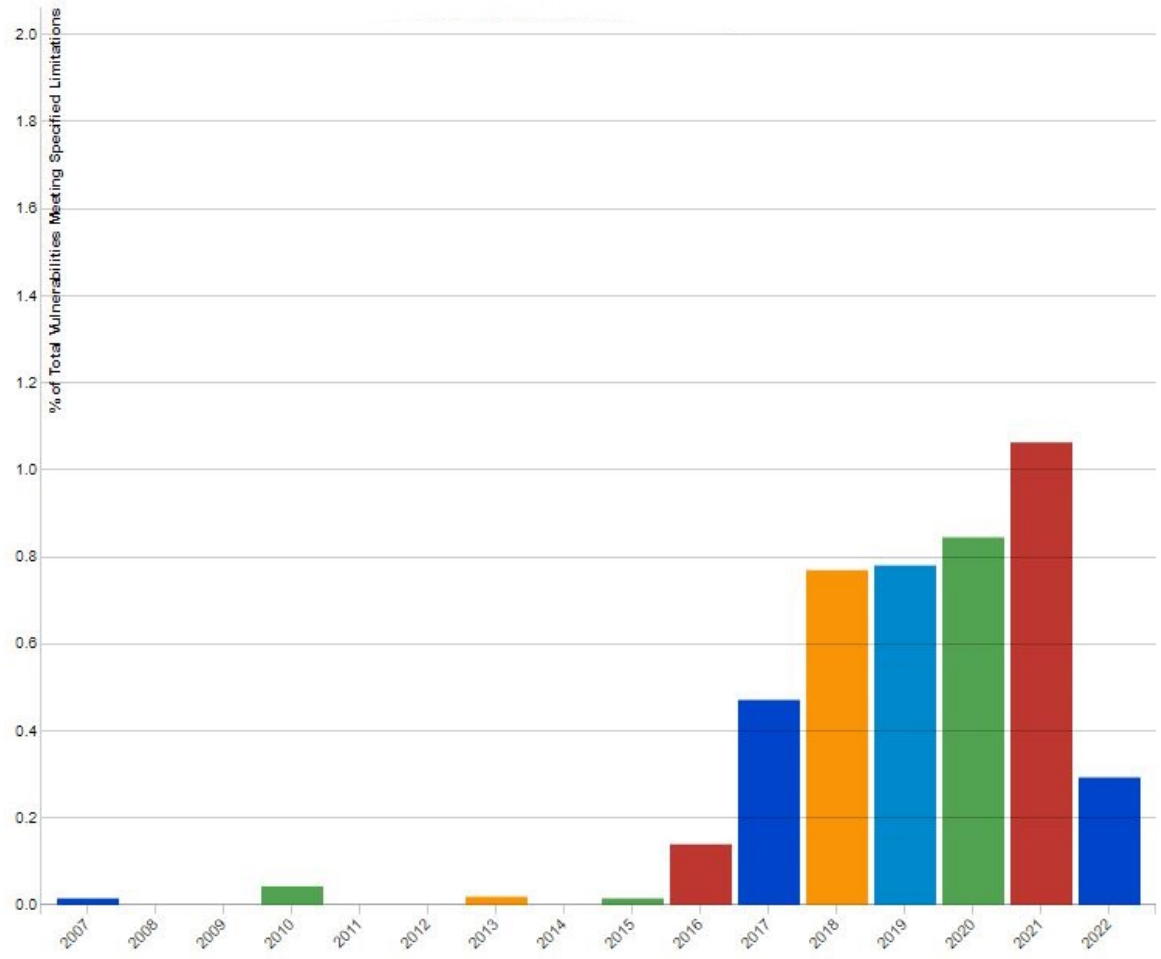

Figure 2.1: Percentage of CWE-502 Deserialization of Untrusted Data (data taken on February 2022).

after risk rating.

In fact, the major reason for the presence of this vulnerability in the Top 10 is that the impact of deserialization flaws cannot be overstated. Because these flaws can lead to remote code execution attacks, one of the most serious attacks possible. Another reason is the prevalence and detectability of the Insecure Deserialization that has a medium risk, so lower than the impact risk, but still remarkable because some tools can discover deserialization flaws, but human assistance is frequently needed to validate the problem. While, the exploitation of deserialization is somewhat difficult, as off the shelf exploits rarely work without changes or tweaks to the underlying exploit code [19].

In Table 2.1 the matches represent the CVE founded in that year, while total are the total number of vulnerabilities discovered in that year. Percentage is trivially the ratio of matches over total, thus giving information on the presence of Deserialization of Untrusted

Year	Matches	Total	Percentage
2007	1	6516	0,02%
2008	0	5632	0,00%
2009	0	5732	0,00%
2010	2	4639	0,04%
2011	0	4150	0,00%
2012	0	5288	0,00%
2013	1	5187	0,02%
2014	0	7937	0,00%
2015	1	6487	0,02%
2016	9	6447	0,14%
2017	69	14645	0,47%
2018	127	16509	0,77%
2019	135	17305	0,78%
2020	155	18351	0,84%
2021	214	20136	1,06%
2022	8	2729	0,29%

Table 2.1: Raw data of CWE-502 Deserialization of Untrusted Data (data taken on February 2022).

Data with respect to all the vulnerabilities present. The latter can be also visible from Figure 2.1 in a graphic way.

2.1.1. Definition

First, it is important to understand what is **serialization** and **deserialization** (see Figure 2.2 for a graphic interpretation).

- *Serialization* is the process of turning some object into a data format that can be restored later.
- *Deserialization* is the reverse of that process, taking data structured from some format, and rebuilding it into an object.

Today, the most popular data format for serializing data is JSON. Before that, it was XML.

However, many programming languages offer a native capability for serializing objects.

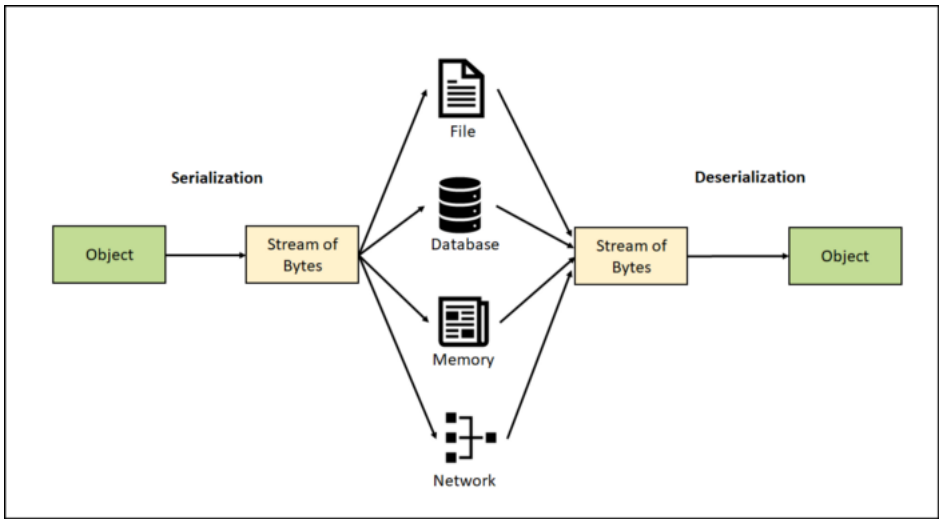

Figure 2.2: Serialization and Deserialization process [12]

These native formats usually offer more features than JSON or XML, including customizability of the serialization process.

Unfortunately, the features of these native deserialization mechanisms can be repurposed for malicious effect when operating on untrusted data. Attacks against deserializers have been found to allow denial-of-service, access control, and remote code execution (RCE) attacks. [17]

2.1.2. Identification and Attack Surface Estimation

This section gives some hints on how to identify Insecure Deserialization vulnerability and its attack surface. The work of this thesis is mostly focused on the Java programming language. In this section will be tackle both the white box approach and the black box approach in Java even if this thesis is mainly focused on the static analysis and so the white box approach.
The distinction between the two approaches is that the first one, the **white box** approach, is a way of testing the software in which the tester has knowledge about the internal structure or the code or the program of the software, while the second one, the **black box** approach, is a way of software testing in which the internal structure or the program or the code is hidden and nothing is known about it.
Therefore, after understanding these basic concepts, it is possible to analyze the two different methods [17]:

- *White box*: any class that implements the interface java.io.Serializable can be serialized and deserialized. If you have source code access, take note of any code that

uses the *readObject()* method, which is used to read and deserialize data from an *InputStream*. Furthermore:

1. *XMLdecoder* with external user defined parameters;
2. *XStream* with *fromXML* method (xstream version <= v1.46 is vulnerable to the serialization issue);
3. *ObjectInputStream* with *readObject*;
4. Uses of *readObject*, *readObjectNodData*, *readResolve* or *readExternal*;
5. *ObjectInputStream.readUnshared*;
6. *Serializable*;
7. *readObject()* method of the java.beans.XMLDecoder class;

- *Black box*: serialized Java objects always begin with the same bytes, which are encoded as ac ed in hexadecimal and rO0 in Base64.

The second part of this section will provide some usages of Java serialization, also called attack surface [15]:

- Servlets HTTP, Sockets, Session Manager, Protocols: RMI (Remote Method Invocation), JMX (Java Management Extension), JMS (Java Messaging System), JNDI, Clusters, Spring Service Invokers (HTTP, JMS, RMI, etc..), etc..
- HTTP Params (ViewState), Cookies, Ajax Components, etc..
- XML (XStream), JSON (Jackson)
- Android, AMF (Action Message Format), JSF ViewState, WebLogic T3, etc..

2.1.3. Exploitation and Vulnerability example

In this section we present how to exploit this vulnerability in order to gain some privileges or unexpected behaviors. Exploitation requires a chain of serialized objects triggering interesting functionality (e.g., writing files, dynamic method calls using Java's reflection API, etc.). For such a chain the term "gadget" got established. Furthermore, the exploitation of the Insecure Deseserialization vulnerability, as previously stated, is somewhat difficult because off the shelf exploits rarely work without changes. However, there are some tools that help you in creating the exploit and one of the most important is called **Ysoserial** [9].
Ysoserial is a collection of utilities and property-oriented programming "gadget chains"

discovered in common java libraries that can, under the right conditions, exploit Java applications performing unsafe deserialization of objects. The main driver program takes a user-specified command and wraps it in the user-specified gadget chain, then serializes these objects to stdout. When an application with the required gadgets on the classpath unsafely deserializes this data, the chain will automatically be invoked and cause the command to be executed on the application host. It should be noted that the vulnerability lies in the application performing unsafe deserialization and not in having gadgets on the classpath. To better understand the concept, it will be shown below an example of a vulnerable web application to the Insecure Deserialization vulnerability [29]. The entire code of this example can be found in Appendix A, while here we show just some snippets of code relevant to the understanding of the general functioning. This web application has three classes:

The first is the **Server** class used mostly to set up the server (see Listing 1 for the entire code).

```
1 Tomcat tomcat = new Tomcat();
2 ...
3 tomcat.start();
4 tomcat.getServer().await();
```

Then, the **Serial** class (Listing 2 for the complete code) is the vulnerable class in which is present the Insecure Deserialization vulnerability and, in particular, in the method *fromBase64()* at rows 3 and 4 of the below snippet. It opens an *ObjectInputStream* and across this receives a deserialized object using the *readObject* method. Function calls defined inside *readObject* generally operate on data read from the stream, and such data can be controlled by an attacker. In such a context, an attacker can craft nested class objects in the deserialization input stream and define a sequence of method calls that end up executing dangerous operations at the operating system level, such as filesystem activities, command execution, etc.

```
1 public static Object fromBase64(String s) throws Exception {
2         byte[] data = new Base64().decode(s);
3         ObjectInputStream ois = new ObjectInputStream(new
          ↪  ByteArrayInputStream(data));
4         Object o = ois.readObject();
5         ois.close();
6         return o;
7 }
```

Finally, the last class is called **Servlet** (Listing 3 for the complete code) and is the one that calls the vulnerable method of the class Serial inside the method *doPost()* (Listing 2).

```
String data = req.getParameter("data");
if (data == null) {
        data = Serial.toBase64(new String("text"));
} else {
        try {
                Serial.fromBase64(data);
        } catch (ClassNotFoundException e) {...}
}
```

Thus, this application will attempt to java deserialize user provided input. Since commons-collections4:4.0 is on the classpath, it can be used for playing around with exploitation. In fact, an example attack using the tool Ysoserial is the following one:

```
java -jar ysoserial-0.0.4-all.jar CommonsCollections4 'shell command...'
| base64 | tr -d "\n"
```

This command will create a payload that sent to the vulnerable program will execute the shell command written. This can lead also to RCE that is one of the most dangerous vulnerabilities because it allows an attacker to remotely run malicious code within the target system on the local network or over the Internet. A RCE vulnerability can lead to loss of control over the system or its individual components, as well as theft of sensitive data.

In summary, three constraints need to be satisfied in order to obtain a successful attack on a Java application: (1) the attacker needs to define his own invocation sequence by starting from a serializable class that redefines *readObject*; (2) to obtain malicious behavior, the attacker has to find a path that starts from the deserialized class and reaches the invocation of one or more desired methods; (3) all the classes considered in the attack execution path must be present in the application's classpath.

2.1.4. Impacts and Mitigations

Impacts

An interesting paper of Ian Haken analyzes the deserialization vulnerability and, using the Gadget Inspector tool (a Java bytecode analysis tool for finding gadget chains), finds out some vulnerable libraries [11]. The work consists in running Gadget Inspector against the

100 most popular java libraries looking for exploits against standard Java deserialization. The results are:

- commons-collections » commons-collections allows RCE (38th most popular maven dependency)
- org.clojure » clojure allows RCE (6th most popular maven dependency)
- org.scala-lang » scala-library allows you to write/overwrite a file with 0 bytes. Possible DoS? Zero-out a blacklist? (3rd most popular maven dependency)
- more vulnerable libraries can be found consulting Ysoserial [9]

Mitigations and Prevention

There are some mitigations that try to solve this problem. The best solution should be to not accept serialized objects from untrusted sources or to use serialization mediums that only permit primitive data types. However, this is not always possible, and some guides [5, 17–19] present intermediate solutions:

- Avoiding native (de)serialization formats will result to a great reduction of risk. By switching to a pure data format like JSON or XML, you lessen the chance of custom deserialization logic being repurposed towards malicious ends.
- Make fields transient to protect them from deserialization. An attempt to serialize and then deserialize a class containing transient fields will result in NULLs where the transient data should be. This is an excellent way to prevent time, environment-based, or sensitive variables from being carried over and used improperly.
- Some application objects may be forced to implement Serializable due to their hierarchy. To guarantee that your application objects cannot be deserialized, a *readObject()* method should be declared (with a final modifier) which always throws an exception.
- The general idea is to override ObjectInputStream.html#resolveClass() in order to restrict which classes are allowed to be deserialized.
- Use a Java agent to override the internals of *ObjectInputStream* to prevent exploitation of known dangerous types (for example contrast-rO0 [6] or NotSoSerial [13]). But, it is noteworthy that allow listing is safer than deny listing.
- If available, use the signing/sealing features of the programming language to assure that deserialized data has not been tainted. For example, a hash-based message

authentication code (HMAC) could be used to ensure that data has not been modified.

- Authenticate prior to deserializing.
- When deserializing data, populate a new object rather than just deserializing. The result is that the data flows through safe input validation and that the functions are safe.
- Replace your deserialization *ObjectInputStream* with SerialKiller [4] or use a safe replacement for the generic *readObject()* method.
- SWAT (Serial Whitelist Application Trainer) [25] helps you to build a whitelist for classic Java deserialization occurrences as well as XStream based ones.

For other mitigations please consult the bibliography [5, 17–19].

2.2. Static Analysis

The approach chosen for this thesis is the usage of the **static analysis** rather than the dynamic analysis. First, we describe both the approaches and then the explanation on why we have selected the first one.

One approach is *static analysis*. This is a generic name for a set of program analysis techniques performed without actually executing a program. Usually, static analysis in software engineering is performed on source code. But, in some cases, source code is often unavailable and, so, it should resort to analysis of binary (i.e., machine) code. In our context, we have the source code and so we do not need to resort to the analysis of the binary. Thus, static techniques analyze unchanging information such as the source code of the software or the one of its environment. They are helpful for modeling control-flow information as well as data-dependency information.

The second technique is *dynamic analysis*. It is based on running executables in an instrumented environment. In other words, with dynamic analysis it is possible to trace the changes that a program makes to a system.

The advantages of using static techniques are its code coverage and the scalability. In fact, static analysis does not require many resources, and it gives a full view of a program, since it analyzes also portions of code that are not revealed during the execution and that could contain dormant functionality.

In this thesis, we select to choose the static analysis for mainly two reasons:

1. Detect the Insecure Deserialization vulnerability is easier with static analysis than dynamic analysis.
2. Since the aim of this thesis is to create a program to test the tools, the tools selected by us are used for the analysis of the source code because the developer will use the static analysis tools during the code writing and so he will have the source code.

The goal of this project is to have a *de facto* standard test for the tools that wants to recognize the Insecure Deserialization vulnerability. Thus, it can be helpful both for the tool's vendor and for the developers that want to write code trying to minimize the possibility of writing insecure code, because our program can be useful to evaluate the performances of the tools that they use and, in case, prefer one tool to the other.

3 | Targets Selection

For the purpose of evaluate the behavior of the tools, in terms of deserialization vulnerability founded, as a first phase we selected a sample of vulnerable targets to Insecure Deserialization that are representative for all the different types of this vulnerability. In particular, a target is a web application written deliberately with vulnerabilities in the code section. The selection has three mandatory requirements: first, that the program is a web application; second that the web application has at least one Insecure Deserialization vulnerability; finally that the target is written in Java. Thus, all the targets chosen respect these three characteristics.
While, for the second phase, we chose a sample of tools that have been selected to represent the most common products used in the market because open source and free (the second phase will be described in details in Section 4.1).
Then, for each couple of target and tool we have analyzed the behavior in order to evaluate the performance of the tool with different kinds of the same vulnerability. Finally, we have summarized the results in a table (Table 4.1) and from the results we extracted the most difficult pattern to recognize in order to make a program that acts like a benchmark for the tools (Section 3.2). This program presents a lot of Insecure Deserialization vulnerabilities different from each other with the aim to test the scanners as completely as possible.
In this chapter we present the targets that have been selected because characterized by the presence of the Insecure Deserialization vulnerability and, also, the program developed in order to test the tools that is called *BenchStress*.

3.1. Vulnerable Targets

In this section will be presented the vulnerable targets to the Insecure Deserialization used as dataset on which record the tools' performances. The targets are seven in order to put to the test the scanners in many different ways and with different levels of complexity.
In order to keep the document light and, so, not too bulky the code of the vulnerable targets is not present here, but can be found on their GitHub page that will be linked

next to their name. These are the targets used:

1. *Serial Killer* [1]. It is a binary exploitation challenge that was given during the All-Army CyberStakes 4 CTF. The application is a simple note keeper in which you can print all notes or add one and the client-server interaction is made with deserialization.
   This target has been chosen because it is a very simple example thus can be useful to understand if a tool is able to find this simple form of Insecure Deserialization.

2. *Java Deserialize webapp* [29]. This target was already described in Section 2.1.3 where it was presented as an example of vulnerable code with related exploitation. This target was chosen because the vulnerable method is inside a class that is called from another class, therefore, it can be tested the ability to trace back the method calls of the tools. But in terms of difficulty is still easy to detect.

3. *DeserLab* [3]. Simple Java client and server application that implements a custom network protocol using the Java serialization format to demonstrate Java deserialization vulnerabilities.
   This program was chosen because it is a bit more complicated with respect to the two before and so it can make harder the reconnaissance by part of the tools.

4. *JavaDeserH2HC* [8]. The lab contains code samples that help you understand deserialization vulnerabilities and how gadget chains exploit them. The goal is to provide a better understanding so that you can develop new payloads and/or better design your environments.
   The complexity of this target raise a bit because there are more class and more vulnerabilities. This can be useful to understand the behavior of the tools when they have to face more vulnerabilities.

5. *WebGoat* [21]. WebGoat is a deliberately insecure application that allows interested developers to test vulnerabilities commonly found in Java-based applications that use common and popular open source components.
   This target is still simple, but the vulnerability is written in a different way and, so, this is a very useful target to test the tools. Because in this way we can test the tools when dealing to the same vulnerability but written in a different mode.

6. *CVE-2017-7525-Jackson-Deserialization-Lab* [14]. Basic Java REST application vulnerable to Insecure Deserialization, leading to RCE.
   This web application target comes in handy when we want to test the tools with the Insecure Deserialization vulnerability because it is not used the usual form of

deserialization adopted by the targets above, but it is used another way, less known, that is still vulnerable to deserialization.

7. *Java Deserialization Of Untrusted Data PoC* [7]. Here there are practical examples of the - deserialization of untrusted data - vulnerability. These pocs use the ysoserial tool to generate exploits.
   This last target is different from the others above because it implements a slightly different way of Insecure Deserialization vulnerability. Moreover, it leverages also on other platform like JBoss, Jenkins and Bamboo.

3.2. BenchStress

In this section, we introduce our own program, called **BenchStress**, that is a benchmark program developed to test the tools. The code can be found on GitHub [24] and also in Appendix B. However, for convenience, here we present some snippets of code important to understand the functioning and how the vulnerabilities are presented.
First, the program functioning is very simple: it is a note keeper in which you can print all notes or add one and the client-server interaction is made with deserialization.

Thus, we start by introduce the **Client** class. It sets up the socket connection given the server ip and the port:

```
1 socket = new Socket(server_ip, port);
```

Then it starts an infinite *while* loop that will allow three actions:

1. Print all notes
2. Add a new note
3. Exit this program

After, it sets up the streams: ***ObjectOutputStream*** and ***ObjectInputStream***:

```
1 oos = new ObjectOutputStream(socket.getOutputStream());
2 ois = new ObjectInputStream(socket.getInputStream());
```

This is the part in which is used the deserialization to send the object through the streams and it is the insecure part of the code that can be exploited by a malicious person because of this vulnerability.

Finally, there are three *if* for each case, but only the second *if*, in which a new note is created, can be useful for the exploitation:

```
1 String newNoteTitle = scan.nextLine();
2 String newNoteBody = scan.nextLine();
3 Note newNote = new Note(42, newNoteTitle, newNoteBody);
4 oos.writeObject("SAVE");
5 Thread.sleep(100);
6 oos.writeObject(newNote);
```

Because at line 6 of the above snippet of code uses the method ***writeObject()*** that writes the specified object to the *ObjectOutputStream* and so to the server. The Object in this case is a *newNote* that can be overwritten and controlled by us, while there are other *writeObject()* methods used, but they send a predefined string and so cannot be modified from us (for example the one at line 4).

Then, the second class is presented (Appendix B for the entire code). It is called **Note** and it is a really simple one because it has just three attributes, a constructor and only a method that will print a note.

```
1 public class Note implements Serializable{
2     private Integer note_id; private String note_title; private String note_body;
3
4     public void print_note(){
5         System.out.println(note_title + "\n" + note_body + "\n");
6     }
7 }
```

Finally, the last class, called **Server.java**, is introduced. This is the most important class because it is the one that keeps all the information and, so, it should be the safest even if in this case it is not.
For the sake of the exploitation, the Apache commons collections libraries are imported because of their well-known vulnerability (Section 2.1.4) that can be leveraged through the Ysoserial tool [9].

```
1 import org.apache.commons.collections4.*;
2 import org.apache.commons.collections4.CollectionUtils;
```

Referring to the code the Server starts initializing a new *ServerSocket* (line 3) and then, the *socket* will start to accept connections (line 7).

```
ServerSocket server = null;
try {
    server = new ServerSocket(8888);
} catch (IOException e) {
    System.err.println(e.getMessage()); /* port not available*/ return;
}
Socket socket = server.accept();
```

Then, the Server will create an example of notes and a list to store them. After, there is an infinite *while* loop also for the server that is inside of another infinite *while* loop. Inside the nested *while* loop it creates the streams:

```
ObjectInputStream ois = new ObjectInputStream(socket.getInputStream());
ObjectOutputStream oos = new ObjectOutputStream(socket.getOutputStream());
```

Here, it will start the most important part of the program in which are present the Insecure Deserialization vulnerabilities in many different ways in order to test out the tools. As we understood from Chapter 2, the vulnerability itself is in these two lines:

```
ObjectInputStream ois=new ObjectInputStream(socket.getInputStream());
Object o = ois.readObject();
```

Thus, for this reason, we presented these two lines in many different ways in order to make difficult the reconnaissance from the tools point of view.

The first is the following:

```
try (ObjectInputStream ois2 = new ObjectInputStream(socket.getInputStream())) {
    Object o = ois2.readObject();
} catch (Exception e) {
}
```

In these four lines the *ObjectInputStream* is instantiated in the *try-catch* block and for this reason it can be difficult to discover from some tools.

Another possible way to disguise the vulnerability can be:

```
if (val) {
    ObjectInputStream ois3 = new ObjectInputStream(socket.getInputStream());
    Object o3 = ois3.readObject();
}else {
```

```
    ObjectInputStream ois4 = new ObjectInputStream(socket.getInputStream());
    Object o4 = ois4.readObject();
}
```

Here the fact that the vulnerable code is inside the *if* could create problems to some tools. *val* in this case is a boolean value that is always true and for this reason it is possible to evaluate the performances of the tools also from the optimization point of view.

After the *if* condition there is another condition statement called *switch*:

```
switch (valInt) {
    case 1:
        System.out.println(valInt);
        break;
    case 6:
        ObjectInputStream ois5 = new ObjectInputStream(socket.getInputStream());
        Object o5 = ois5.readObject();
        break;
    default:
        break;
}
```

valInt is an int value that is initialized to 6 and so the *switch* should enter only in the second case. As the *if* condition, here we evaluate both the security and the optimization parts of the tool.

Another way to test the tools could be with the *finally* clause of the *try-catch-finally* block:

```
try {
    System.out.println(message);
}catch (Exception e){
    System.err.println(e);
}finally {
    ObjectInputStream ois6 = new ObjectInputStream(socket.getInputStream());
    message = (String) ois6.readObject();
}
```

The fact that the vulnerable code is in the *finally* clause can create issues to some scanners.

Next, the conditional operator is used to challenge the reconnaissance of the vulnerability:

```
1 ObjectInputStream ois6 = null;
2 ois6 = (val) ? new ObjectInputStream(socket.getInputStream()): null;
3 message = (!val) ? null: (String)ois6.readObject();
```

Conditional operator is also known as the ternary operator. This operator consists of three operands and is used to evaluate boolean expressions. The goal of the operator is to decide, which value should be assigned to the variable. Hence, at line 2 of the above code, the *ObjectInputStream ois6*, if *val* is true, is equal to *new ObjectInputStream(socket.getInputStream())* otherwise to *null*.

Another way to test out the tools is with the *for* loop and, in particular, as follows:

```
1 ObjectInputStream[] oisArray = new ObjectInputStream[10];
2 for (i=0; i<10; oisArray[i] = new ObjectInputStream(socket.getInputStream())) {
3     Object o= oisArray[i].readObject();
4     i++;
5 }
```

Here, the code is a bit tricky, but it can put in serious trouble many tools. First, it creates an array of *ObjectInputStream*, then iterate a loop by incrementing the index *i* and, instead of putting the incrementation in the for statement, as it is usually done, it is replaced by the instantiation of a new *ObjectInputStream*.

At the end, there is a control flow statement called *do-while* loop:

```
1 do {
2     ObjectInputStream ois8 = new ObjectInputStream(socket.getInputStream());
3     message = (String) ois8.readObject();
4     i+=10;
5 }while (i<100);
```

In the lines above the vulnerable part is inside the loop and this can create problems sometimes.

Finally, there is an *if-else* condition that is useful for the functioning of the program:

```
1 if(message.equalsIgnoreCase("GET")){
2     // Client wants all of the saved notes
3     oos.writeObject(noteList);
4     continue;
5 }else if (message.equalsIgnoreCase("SAVE")){
```

```
6       // Client wants to save a note, accept a Note object
7       Note newNote = (Note) ois.readObject();
8       noteList.add(newNote);
9       continue;
10  }else if (message.equalsIgnoreCase("BYE")){
11      break;
12  }else{
13  }
```

In which, if the message sent from the client is a "GET", then the entire *noteList* will be sent to the client in which all the notes are saved. While, if the client has sent a "SAVE" message, then a new note is added to the list and here is present another vulnerability of the Server class. At the end, if the message is "BYE", it will exit from the loop and terminate.

4 | Experimental Results

In this chapter we first present the tools used in this thesis to analyze the code of the target web applications. In particular, the description of the tools on how they work and how they recognize the vulnerabilities in the code.
The tools, as mentioned in the previous chapter, are selected in a way to represent the most common products used in the market because are user-friendly, free and open source. Moreover, another aspect that we have taken in consideration during the selection of the tools is that they presents the ability to recognize the Insecure Deserialization vulnerability. This can be proven by the presence of rules, the so called bug patterns that will be described below, containing the Insecure Deserialization vulnerability.
Next, we present the experiment with its relative results and, furthermore, the interpretation of the results. Our main aim is to build a program for improving the automatic discovery and recognition of Insecure Deserialization vulnerabilities by the tools.

All the results will be summarized in a table (Table 4.1) and the description of the results more in details will follow the table.

4.1. List of Analyzed Tools

The first step consists in the selection of the tools. Since the aim of this thesis is the static analysis of the code, the choice falls on the **Static Application Security Testing (SAST) tools** because they are used to secure software by reviewing the source code of the software to identify sources of vulnerabilities.
Unlike dynamic application security testing (DAST) tools for black-box testing of application functionality, SAST tools focus on the code content of the application, white-box testing. The reason why the analysis of the source code is preferred it is because the Insecure Deserialization vulnerability is easier to find out through the code analysis, as can be understood in the Chapter 2.
Next, it will be introduced all the tools used for the analysis. Furthermore, for each tool will be present also a subsection in which is analyzed the complete list of descriptions given when the tool identifies potential weaknesses that is called "**Bug Patterns**". Ba-

sically, the Bug Patterns, as the name suggest, are the patterns and so what is searched by the tool in order to recognize the bug. Since the focus of this thesis is on the Insecure Deserialization vulnerability, among all the patterns only the ones related with the Object Deserialization will be addressed. The list is the following:

1. **Find Security Bugs** (Section 4.1.1);
2. **SonarLint** (Section 4.1.2);
3. **Error Prone** (Section 4.1.3);
4. **SonarQube** (Section 4.1.4);
5. **Semgrep** (Section 4.1.5);

We now introduce all the tools in details one by one.

4.1.1. Find Security Bugs

Find Security Bugs is the SpotBugs plugin for security audits of Java web applications [2].

Bug Patterns

This subsection, as previously stated, analyzes the bug patterns related to the deserialization bug.

1. *OBJECT_DESERIALIZATION*. Object deserialization is used and code at risk:
   Here the pattern that is searched by the tool is *readObject()*, that, as already stated in the previous chapter, is the most common form of deserialization used from the devolepers. This is very dangerous because it can lead to remote code execution.

```
1    public UserData deserializeObject(InputStream receivedFile) throws
     ↪  IOException, ClassNotFoundException {
2
3    try (ObjectInputStream in = new ObjectInputStream(receivedFile)) {
4        return (UserData) in.readObject();
5        }
6    }
```

2. *JACKSON_UNSAFE_DESERIALIZATION*. Unsafe Jackson deserialization configuration and code at risk:
   The main trigger that creates an alert is the presence of the method *enableDefaultTyping()*. The second example, instead, shows that also the presence of *readValue()*

can lead to deserialization and, so, lead to remote code execution.

```
public void example(String json) throws JsonMappingException {
     ObjectMapper mapper = new ObjectMapper();
     mapper.enableDefaultTyping();
     mapper.readValue(json, ABean.class);
}

public void exampleTwo(String json) throws JsonMappingException {
     ObjectMapper mapper = new ObjectMapper();
     mapper.readValue(json, AnotherBean.class);
}
```

3. *DESERIALIZATION_GADGET*. This class could be used as deserialization gadget. Deserialization gadget are class that could be used by an attacker to take advantage of a remote API using Native Serialization. This class is either adding custom behavior to deserialization with the *readObject* method (*Serializable*) or can be called from a serialized object (*InvocationHandler*).

4. *XML_DECODER*. *XMLDecoder* should not be used to parse untrusted data. Deserializing user input can lead to arbitrary code execution. Vulnerable Code:
This pattern still looks for the *readObject()* method (like the first bug pattern), although in this case it is not called from the *ObjectInputStream* class but from the *XMLDecoder* class.

```
XMLDecoder d = new XMLDecoder(in);
try {
    Object result = d.readObject();
}
```

5. *LDAP_ENTRY_POISONING*. JNDI API support the binding of serialize object in LDAP directories. Vulnerable code:
The exploitation of the vulnerability will be possible if the attacker has an entry point in an LDAP base query, by adding attributes to an existing LDAP entry or by configuring the application to use a malicious LDAP server. The deserialization is possible when the fifth parameter of the constructor of *SearchControls* is true because it returns the object bound to the name of the entry; if false, instead, does not return object avoiding the deserialization problem.

```
DirContext ctx = new InitialDirContext();
// ...
ctx.search(query, filter,
        new SearchControls(scope, countLimit, timeLimit, attributes,
            true, //Enable object deserialization if bound in directory
            deref));
```

6. *RPC_ENABLED_EXTENSIONS*. Enabling extensions in Apache XML RPC server or client can lead to deserialization vulnerability which would allow an attacker to execute arbitrary code. It is recommended not to use *setEnabledForExtensions* method of *org.apache.xmlrpc.client.XmlRpcClientConfigImpl* or, also, from the *org.apache.xmlrpc.XmlRpcConfigImpl*. By default, extensions are disabled both on the client and the server.

For more details please consult the website of SpotBugs [2].

4.1.2. SonarLint

SonarLint is a free and open source IDE extension that identifies and helps you fix quality and security issues as you code. Like a spell checker, SonarLint squiggles flaws and provides real-time feedback and clear remediation guidance to deliver clean code from the get-go [26].

Bug Patterns

This subsection, as previously stated, analyzes the bug patterns related to the deserialization bug.

1. *Deserialization should not be vulnerable to injection attacks*. Noncompliant Code Example:
   The bug pattern of this tool is very similar to the first one adopted from the tool described above. In fact, here the pattern that is searched by the tool is the *readObject()* method, that it is the most common way to use deserialization in Java.

```
public class RequestProcessor {
  protected void processRequest(HttpServletRequest request) {
    ServletInputStream sis = request.getInputStream();
    ObjectInputStream ois = new ObjectInputStream(sis);
    Object obj = ois.readObject(); // Noncompliant
```

```
    }
}
```

2. *Using unsafe Jackson deserialization configuration is security-sensitive.* Sensitive code example:
   This rule raises an issue when: *enableDefaultTyping()* is called on an instance of *com.fasterxml.jackson.databind.ObjectMapper* or, also, from an instance of the library *org.codehaus.jackson.map.ObjectMapper*; otherwise, when the annotation *@JsonTypeInfo* is set at class, interface or field levels and configured with use = *JsonTypeInfo.Id.CLASS* or use = *Id.MINIMAL_CLASS*. This pattern, as the previous one, is very similar to the bug pattern used from SpotBugs, in particular the second one.

```
ObjectMapper mapper = new ObjectMapper();
mapper.enableDefaultTyping(); // Sensitive
// or another example
@JsonTypeInfo(use = Id.CLASS) // Sensitive
abstract class PhoneNumber {
}
```

3. *Allowing deserialization of LDAP objects is security-sensitive.* Sensitive Code Example:
   This rule raises an issue when an LDAP search query is executed with *SearchControls* configured to allow deserialization. It is interesting to note that, like the previous ones, also this bug pattern used by SonarLint is very similar to one that is adopted by SpotBugs (the fifth one).

```
DirContext ctx = new InitialDirContext();
// ...
ctx.search(query, filter,
        new SearchControls(scope, countLimit, timeLimit, attributes,
            true, // Noncompliant; allows deserialization
            deref));
```

4. *"ActiveMQConnectionFactory" should not be vulnerable to malicious code deserialization.* Noncompliant Code Example:
   ActiveMQ can send/receive JMS Object messages (named *ObjectMessage* in *ActiveMQ* context) to comply with JMS specification. Internally, *ActiveMQ* relies on

Java serialization mechanism for marshaling/unmarshaling of the message payload. Deserialization based on data supplied by the user could lead to remote code execution attacks, where the structure of the serialized data is changed to modify the behavior of the object being unserialized.

```
1    ActiveMQConnectionFactory factory = new
     ↪  ActiveMQConnectionFactory("tcp://localhost:61616");
2    factory.setTrustAllPackages(true); // Noncompliant
3
4    ActiveMQConnectionFactory factory = new
     ↪  ActiveMQConnectionFactory("tcp://localhost:61616");
5    // no call to factory.setTrustedPackages(...);
```

For more details please consult the website of SonarSource [28].

4.1.3. Error Prone

Error Prone hooks into your standard build, so all developers run it without thinking, tells you about mistakes immediately after they are made and produces suggested fixes, allowing you to build tooling on it [10].

Bug Patterns

This subsection, as previously stated, analyzes the bug patterns related to the deserialization bug.

1. *BundleDeserializationCast* (Severity ERROR). Object serialized in Bundle may have been flattened to base type.
2. *BanSerializableRead* (Severity ERROR). Deserializing user input via the 'Serializable' API is extremely dangerous.
   The Java *Serializable* API is very powerful, and very dangerous. Any consumption of a serialized object that cannot be explicitly trusted will likely result in a critical remote code execution bug that will give an attacker control of the application.

For more details please consult the website of Error Prone [10].

4.1.4. SonarQube

SonarQube is an open-source platform developed by SonarSource for continuous inspection of code quality to perform automatic reviews with static analysis of code to detect bugs,

code smells and security vulnerabilities on 20+ programming languages. SonarQube offers reports on duplicated code, coding standards, unit tests, code coverage, code complexity, comments, bugs and security vulnerabilities [27].

Bug Patterns

Since the tool belongs to the same company of SonarLint, they use the same rules to discover the vulnerabilities. Thus, please consult Section 4.1.2 of SonarLint. But, even if the tools share the same rules, during the analysis of the tools they had different results and, for this reason, both are present in the thesis.

4.1.5. Semgrep

Semgrep is a fast, open-source, static analysis tool for finding bugs and enforcing code standards at editor, commit and CI time [23].

Bug Patterns

This subsection, as previously stated, analyzes the bug patterns related to the deserialization bug.

1. *java.jax-rs.security.insecure-resteasy.insecure-resteasy-deserialization*
   Test code:
   When a Restful webservice endpoint is configured to use wildcard *mediaType* {*/*} as a value for the @Consumes annotation, an attacker could abuse the *SerializableProvider* by sending a HTTP Request with a Content-Type of application/x-java-serialized-object. The body of that request would be processed by the *SerializationProvider* and could contain a malicious payload, which may lead to arbitrary code execution when calling the *$Y.getObject* method.

```
1    @Path("/")
2    public class PoC_resource {
3        @POST
4        @Path("/concat")
5        @Produces(MediaType.APPLICATION_JSON)
6        // ruleid: insecure-resteasy-deserialization
7        @Consumes({"*/*"})
8        public Map<String, String> doConcat(Pair pair) {...}
9    }
```

2. *java.lang.security.audit.object-deserialization.object-deserialization*
   Test code:
   This is very similar to the patterns written in the previous tools, but there is one main difference with respect to the other ones that is the pattern searched from Semgrep, because it is not *readObject()* but it is *new ObjectInputStream()*.

```
    public UserData deserializeObject(InputStream receivedFile) throws
    ↪  IOException, ClassNotFoundException {
        // ruleid:object-deserialization
        ObjectInputStream in = new ObjectInputStream(receivedFile);
        return (UserData) in.readObject();
    }
```

3. *java.lang.security.insecure-jms-deserialization.insecure-jms-deserialization*
   Test code:
   JMS Object messages depend on Java Serialization for marshalling/unmarshalling of the message payload when *ObjectMessage.getObject()* is called. Thus, the pattern that will trigger the tool is the *getObject()* method.

```
    ObjectMessage msg = (ObjectMessage) message;

    // ruleid: insecure-jms-deserialization
    Object o = msg.getObject(); //variant 1: calling getObject method directly
    ↪  on an ObjectMessage object

    // ruleid: insecure-jms-deserialization
    Income income = (Income) msg.getObject(); //variant 2: calling getObject
    ↪  method and casting to a custom class
```

4. *java.lang.security.use-snakeyaml-constructor.use-snakeyaml-constructor*
   Test code:
   Used SnakeYAML *org.yaml.snakeyaml.Yaml()* constructor with no arguments, which is vulnerable to deserialization attacks. The pattern that will trigger the tool is when it is instiantiated a new YAML object: *new org.yaml.snakeyaml.Yaml()*.

```
    public void unsafeLoad(String toLoad) {
        // ruleid:use-snakeyaml-constructor
        Yaml yaml = new Yaml();
        yaml.load(toLoad);
    }
```

5. *java.rmi.security.server-dangerous-class-deserialization.server-dangerous-class-deserialization*
   Test code:
   Using a non-primitive class with Java RMI may be an insecure deserialization vulnerability. Depending on the underlying implementation. This object could be manipulated by a malicious actor allowing them to execute code on your system.

```
1    // ruleid:server-dangerous-class-deserialization
2    public interface IBSidesService extends Remote {
3       boolean registerTicket(String ticketID) throws RemoteException;
4       void vistTalk(String talkname) throws RemoteException;
5       void poke(Attendee attende) throws RemoteException;
6    }
```

6. *java.rmi.security.server-dangerous-object-deserialization.server-dangerous-object-deserialization*
   Test code:
   It is interesting to note that this bug pattern is very similar to the one above, except for the difference that in the method *poke()* the parameter required in the first case is of type *Attendee*, while here the parameter is an *Object*. Hence, this is a more general case with respect to the previous one.

```
1    // ruleid:server-dangerous-object-deserialization
2    public interface IBSidesService extends Remote {
3       boolean registerTicket(String ticketID) throws RemoteException;
4       void vistTalk(String talkname) throws RemoteException;
5       void poke(Object attende) throws RemoteException;
6    }
```

7. *java.lang.security.audit.xml-decoder.xml-decoder*
   Test code:
   This bug pattern is very similar to the fourth of SpotBugs, but there is one difference that is the pattern searched from the tool. Because in the previous case SpotBugs was looking for the *readObject()* method, while Semgrep is looking for the instantiation of a *new XMLDecoder()*.

```
// ruleid: xml-decoder
public static Object handleXml(InputStream in) {
    XMLDecoder d = new XMLDecoder(in);
    try {
        Object result = d.readObject(); //Deserialization happen here
    }
}
```

8. *mobsf.mobsfscan.jackson_ deserialization.jackson_ deserialization*
   Pattern:
   The app uses jackson deserialization library and, in particular, the tool is looking for an instance of *com.fasterxml.jackson.databind.ObjectMapper* when it calls the method *enableDefaultTyping()*. Also, this bug pattern is very similar to other bug patterns of the tools described above.

```
import com.fasterxml.jackson.databind.ObjectMapper;
...
$Z.enableDefaultTyping();
```

9. *mobsf.mobsfscan.object_ deserialization.object_ deserialization*
   Pattern:
   This pattern is very simple and find object deserialization using *ObjectInputStream*. In fact, it searches for the string *new ObjectInputStream()*.

```
new ObjectInputStream(...);
```

10. *mobsf.mobsfscan.xmldecoder_ xxe.xml_ decoder_ xxe*
    Pattern:
    XMLDecoder should not be used to parse untrusted data. Deserializing user input can lead to arbitrary code execution. For this reason, the tool is looking for the pattern *new XMLDecoder()*.

```
$X $METHOD(...) {
    ...
    new XMLDecoder(...);
    ...
}
```

For more details please consult the website of Semgrep [23].

4.2. Experiment

The experiment consists in five different tools (Section 4.1) launched over seven different vulnerable targets (Section 3.1) and, in addition, the one that we have implemented (Section 3.2). The goal of this analysis is to figure out the behavior of the tools and their performances.

The main parameter of the analysis that is taken in consideration is only if the tool is able to recognize all the Insecure Deserialization vulnerabilities and if not, the number of vulnerabilities found. Thus, in this thesis we do not consider the optimization of the tools, for example the time needed to analyze all the code, neither their performances with respect to other vulnerabilities.

Another interesting performance metric for the evaluation phase could be the precision and recall (Figure 4.1).

First, it is important to understand what the four values mean:

- True Positive (TP): A test result that correctly indicates the presence of a condition or characteristic.
- True Negative (TN): A test result that correctly indicates the absence of a condition or characteristic.
- False Positive (FP): A test result which wrongly indicates that a particular condition or attribute is present.
- False Negative (FN): A test result which wrongly indicates that a particular condition or attribute is absent.

In a classification task, the precision for a class is the number of true positives (i.e. the number of items correctly labelled as belonging to the positive class) divided by the total number of elements labelled as belonging to the positive class (i.e. the sum of true positives and false positives, which are items incorrectly labelled as belonging to the class). Recall in this context is defined as the number of true positives divided by the total number of elements that actually belong to the positive class (i.e. the sum of true positives and false negatives, which are items which were not labelled as belonging to the positive class but should have been) [22]. To summarize in a formula, precision Equation (4.1) and recall

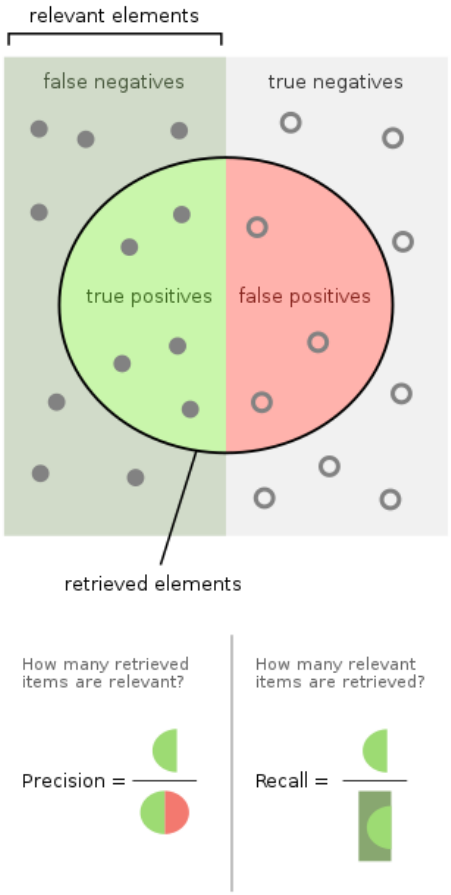

Figure 4.1: Precision and Recall [22].

Equation (4.2) are then defined as:

$$Precision = \frac{tp}{tp + fp} \tag{4.1}$$

$$Recall = \frac{tp}{tp + fn} \tag{4.2}$$

From this we could extract some significant information regarding the tools. For example, the precision can be used as metrics to verify if the vulnerabilities found are true vulnerabilities, while the recall can help to evaluate how many vulnerabilities is able to find overall.

All tests were performed using a build of OpenJDK 8 with Java 8 and, in particular, the version 8u272-b10 for the operating system Windows running on architecture x86 64-bit. The tests were run on a two-core, Intel Core i5-6200U machine with 12 GB of RAM running Windows 10. Moreover, all the tools, apart from Semgrep, were launched inside the IDE IntelliJ IDEA Community Edition 2021.2.3. Semgrep instead was launched via

WebApp\Tool	Find Security Bugs	SonarLint	Error Prone	SonarQube	Semgrep
Serial Killer	Found	Not found	Not found	Not found	Found
Java Deserialize webapp	Found	Not found	Not found	Not found	Found
DeserLab	Found	Not found	Not found	Not found	Found
JavaDeserH2HC	Found	Not found	Not found	Not found	Found
WebGoat	Found	Not found	Not found	Not found	Partially found
CVE-2017-7525-Jackson-Deserialization-Lab	Partially found	Not found	Not found	Partially found	Partially found
Java Deserialization Of Untrusted Data PoC	Found	Not found	Not found	Not found	Found
BenchStress	Partially found	Not found	Not found	Not found	Partially found

Table 4.1: Results of the experiment.

Windows Subsystems for Linux (WSL) running Ubuntu 20.04.3 LTS.

4.3. Results

In Table 4.1 the rows represent the different targets, while the columns are the tools and, as is clearly visible, the intersection is when the tool is launched on that particular target. Then, we distinguish the result in three possible outcome: "Found", in green, means that the tool was able to find all the vulnerabilities present in the target program; "Not found", in red, means that the tool was not able to find any vulnerability in the target; "Partially found", in orange, means that the tool was able to find at least one vulnerability in the target program.

The majority of the tools cannot find any vulnerability, however SpotBugs and Semgrep are able to find almost every vulnerability except a few and in the section below we find out how they works. This tell us that, even if the Insecure Deserialization vulnerability is very well-know from 2018, the majority of the tools has not tried to improve this aspect.

4.4. Interpretation of the Results

Looking at the results it is possible to make two observations, orthogonal between them: on the one hand you can see which tool behaves best and which worse, and, on the other, which are the most critical web applications targets identified.
From the perspective of **tools**, SpotBugs is the best one among the tools chosen, while SonarLint and Error Prone are the two that behave worse. **SpotBugs** is the one that behaves best in all circumstances because for each vulnerable target it finds at least the same number of vulnerabilities as the other tools and it usually finds more vulnerabilities than the other tools. While, as stated above, **SonarLint** and **Error Prone** are the ones that behave worse because in every circumstances they are inferior to the other tools and they are never able to find any vulnerability.
From the point of view of the **targets**, instead, it is interesting to note that the one that is harder to be recognized is CVE-2017-7525-Jackson-Deserialization-Lab. Because, even if, both Semgrep, SonarQube and SpotBugs were able to find the majority of the vulnerabilities of the program, they was not able to find all of them. The second vulnerable web application target which is more critical to recognize is OWASP Webgoat because only one tool (SpotBugs) was able to find all the vulnerabilities present, while the other tool (Semgrep) that usually performs well was not able to find all the vulnerabilities present in the program. What is not detected by the tool is:

```
1 try (ObjectInputStream ois = new ObjectInputStream(new
  ↪  ByteArrayInputStream(Base64.getDecoder().decode(b64token)))) {
2     Object o = ois.readObject();
3     if (!(o instanceof VulnerableTaskHolder)) {
4         //Do something
5     }
6 } catch (Exception e) {
7     return failed(this).feedback("insecure-deserialization.invalidversion").build();
8 }
```

Since the vulnerable part is present inside the round parenthesis of the *try* block, it makes difficult the reconnaissance by the tool.
An interesting fact to note is how SpotBugs and Semgrep work, because they work in a different way and, in particular, the difference is in what they look for in a program. The first looks for the method ***readObject()***, while the second looks for the instantiation of a ***new ObjectInputStream***. If the net result of the two "strategies" seems to be rewarding SpotBugs, it is also necessary to keep in mind the differing in terms of speed which shows Semgrep a little bit faster than SpotBugs and other factors, but this is not

	BenchStress	SpotBugs	Semgrep
Vulnerabilities in Client	1	1	1
Vulnerabilities in Server	9	8	7
Overall Vulnerabilities	10	9	8

Table 4.2: Comparison of SpotBugs and Semgrep on the program.

the scope of this thesis. Thus, regarding only the aspect of vulnerabilities found, SpotBugs seems to make the right choice in terms of analysis with respect to SpotBugs because it recognizes more vulnerabilities in almost every situation. In fact, another example that shows the greater efficiency of SpotBugs in comparison to Semgrep is when the two tools are launched on the program that we have implemented (*BenchStress*). Table 4.2 provides general information on the number of vulnerabilities found in the program.

The BenchStress column contains all the vulnerabilities present in the program, while in the columns of SpotBugs and Semgrep there are the number of vulnerabilities found by the tools. Looking simply at the results seems that SpotBugs has found just one more vulnerability than Semgrep, but this is not totally true because the only vulnerability not found by the first tool is placed inside an *else* block that will never satisfy his condition and so it will never reach the code inside the *else*. Hence, for this reason, SpotBugs did not find this vulnerability, or rather it never scan through that code because is unreachable. The following is the code described above:

```
final boolean val=true;
...
if (val) {
    ObjectInputStream ois3 = new ObjectInputStream(socket.getInputStream());
    Object o3 = ois3.readObject();
}else {
    ObjectInputStream ois4 = new ObjectInputStream(socket.getInputStream());  //Found
    ↪  by Semgrep even if this block is unreachable
    Object o4 = ois4.readObject();  //SpotBugs does not find the vulnerability in the
    ↪  else because val is always true
}
```

The fact that SpotBugs does not check the code if not reachable can be an optimization, but this can create some problems: for example in the case if an attacker is able to control the flow of the program and so modify the value of *val*. Even if this is a corner case that consider at least another vulnerability, it is interesting to reflect also about this

because it falls within the argument of precision and recall described above (Section 4.2). For example, in this particular case the classification of the code in the *else* block as vulnerability can be considered like a false positive because it is not reachable and, thus, it will reduce the precision (Equation (4.1)). Hence, the choice of SpotBugs is to maximize the precision, even if this sometimes can reduce the recall because, for example in the code above, if the vulnerability not recognized would have been a real vulnerability it will belong to the false negative thus reducing the recall (Equation (4.2)).
Semgrep, as can be understood from Table 4.2, is not able to detect two vulnerabilities in the program and this can be an interesting point to analyze. The first vulnerability not found by the tool is very similar to the one described above when the OWASP WebGoat application was depicted. In fact, we have taken inspiration also from that program in order to build our own implementation with the aim to test the tools. The code is the following:

```
try (ObjectInputStream ois2 = new ObjectInputStream(socket.getInputStream())) {
    Object o = ois2.readObject();
} catch (Exception e) {
    return;
}
```

This instantiation of a *new ObjectInputStream* inside the *try* round parenthesis creates some problems to the identification of the vulnerability by the tool.
The concept of the second vulnerability not found by Semgrep is similar to the one above but applied in another situation. The code is presented below:

```
ObjectInputStream[] oisArray = new ObjectInputStream[10];
for (i=0; i<10; oisArray[i] = new ObjectInputStream(socket.getInputStream())) {
    Object o= oisArray[i].readObject();
    i++;
}
```

First, an array of *ObjectInputStream* is created in order to store at every iteration a *new ObjectInputStream*. Then, inside the *for* cycle we can find the initialization and the condition as usual, but instead of the standard increment it is replaced by the instantiation of a *new ObjectInputStream*. While, the incrementation is shifted inside the body of the *for* loop. This pattern should trigger the tool and detect the vulnerability, but for some reason Semgrep is not able to detect the vulnerability in this form.

We decided to analyze in detail only these two tools because are the best ones to detect

the Insecure Deserialization vulnerability.

5 | Conclusions and Future Works

This thesis presented a study on SAST tools with respect to Insecure Deserialization vulnerabilities in Java source code. This study was performed on 5 free and open-source SAST tools launched against 7 different targets, with the addition of *BenchStress*, a benchmark application that we created.

Despite the tools analyzed were a small number, we assume that also other tools, even the commercial ones, are based on similar patterns and, also that these may be structurally similar to those analyzed.

Specifically, since we do not have access to the internals of all tools, we studied the behavior and performance of each tool by focusing on their ability to detect vulnerabilities in the testing environment.

First, we have given some background concepts needed to understand this thesis. For instance, the definition of Insecure Deserialization vulnerability and how to identify it. Moreover, a practical example is provided which is useful to understand a real case scenario and, also, how to exploit it. Another concept that is explained is the static analysis and why we use it.

Then, we have presented the targets chosen to test and collect information regarding the tools. The choice of the targets was very crucial because we had to select the web applications well-known to be vulnerable to Insecure Deserialization. Moreover, we have manually confirmed the presence of these vulnerabilities. This was useful to understand if the tool recognizes a real vulnerability (true positive) or it recognize a false vulnerability (false positive).

Finally, the most important part of the thesis, that is also the main part, is the analysis of the results. In which, first we have introduced the tools used for this experiment, with a particular focus on the bug patterns. The bug patterns are the means by which the tools are able or not to detect a particular vulnerability. Next, it is explained the experiment and, then, the results are presented with also some interpretations of the latter. This study highlighted the great difference in terms of performances of different scanners

compared to the same vulnerability. While, some tools were not able to detect almost any vulnerability, others were able to detect almost all the Insecure Deserialization vulnerabilities present in every web application target. Then, we performed a comparison of the two best static security scanners, SpotBugs and Semgrep, when launched on BenchStress, the web application program that we have implemented. Both the tools were capable of detecting the majority of the vulnerabilities, even if Semgrep was the one put in the most trouble from the program because it was not able to recognize two type of Insecure Deserialization because written in a different way.

Despite this facts, there are some positive aspects regarding the problems we studied. They are going to evolve, and in a positive direction. Awareness of information security topics is constantly growing in both the software engineering community and the industry management and it is in the interest of maintainers of programming languages to provide secure technologies to companies and end users in general. At the time of writing, OWASP has included again the Insecure Deserialization from 2017 as a part of a new larger category called *A08:2021-Software and Data Integrity Failures* in the new release of the OWASP Top 10 Web Application Security Risks project for 2021 [20]. Furthermore, also Semgrep has updated the rule for the deserialization and it has added a new pattern that is similar to the one that we used in BenchStress that was not detected by the tool during our analysis, but now it is able to detect it. Thus, the result of Semgrep when launched on our program is improved because now it is able to detect 9 out of 10 Insecure Deserialization vulnerabilities, still one vulnerability is not detected by the scanner. It is important that such directions are clearly imparted and not underestimated, as education of the developers' community is crucial for software quality and security. As most areas of information security, software protection is constantly improving, as new techniques emerge for discovering and treating vulnerabilities. With software codebases becoming bigger and more complex everyday, automated and semi-automated approaches are already paramount in software validation for security. This trend of increasing importance of such techniques gives a clear direction to research, which will keep shifting from manual to automated analysis.

While trying to select tools that are representative of what the market offers, the most obvious limitation of this work lies in the fact that we used only freely available tools and, often, also open source. Thus, we have not tested any commercial security scanner.

The approach based on static analysis suffers from one limitation: when there is a vulnerability at runtime. In such a context, static analysis fails to detect some attack vectors when, for example, the attacker uses reflection or when he is able to dynamically load classes at runtime.

However, this is not a limitation of our technique, but is more generally a limitation of static analysis.

During the development of the thesis, some ideas about possible ways to extend the work came up. We report the following:

- Perform a similar study but with the black-box web vulnerability scanners with respect to the same vulnerability.
- Another interesting work could be to see the performances of the tools in terms of optimization. For example, the time needed to execute the scan.
- Evaluate the metrics of precision and recall could be an extended work of this thesis. Thus, consider the performance also in terms of true positives and false positives.
- Perform a similar study on SAST tools with respect to others important vulnerabilities on OWASP Top 10 list, for example SQL Injections or XXE.
- An addition to this work could be to repeat the same experiment but with a different programming language (e.g., PHP or Python).

Bibliography

[1] All-Army CyberStakes 4. Serial killer aacs 4 ctf, 2020. URL `https://github.com/archang31/aacs4-writeups/tree/master/BinaryExploitation/SerialKiller`.

[2] P. Arteau. Find security bugs, 2020. URL `https://find-sec-bugs.github.io/`.

[3] N. Bloor. Attacking java deserialization, 2017. URL `https://nickbloor.co.uk/2017/08/13/attacking-java-deserialization/`.

[4] L. Carettoni. Defending against Java Deserialization Vulnerabilities. *Hackers to Hackers Conference 2017 (H2HC 2017)*, 2017.

[5] CLASP. Cwe-502: Deserialization of untrusted data, 2006. URL `https://cwe.mitre.org/data/definitions/502.html`.

[6] Contrast-Security-OSS. Contrast-security-oss/contrast-ro0: A tiny java agent that blocks attacks against unsafe deserialization, 2015. URL `https://github.com/Contrast-Security-OSS/contrast-rO0`.

[7] S. Cristalli, E. Vignati, D. Bruschi, and A. Lanzi. Trusted execution path for protecting java applications against deserialization of untrusted data. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 445–464. Springer, 2018.

[8] J. F. M. Figueiredo. An Overview of Deserialization Vulnerabilities in the Java Virtual Machine (JVM). *Hackers to Hackers Conference 2017 (H2HC 2017)*, 2017.

[9] C. Frohoff and G. Lawrence. Marshalling pickles: How deserializing objects will ruin your day. In *OWASP AppSec California*. OWASP, 2015. URL `http://frohoff.github.io/appseccali-marshalling-pickles/`.

[10] Google. Error prone, 2022. URL `https://errorprone.info/`.

[11] I. Haken. Automated discovery of deserialization gadget chains. *Proceedings of the Black Hat USA*, 2018.

[12] A. Issac. Serialization Filtering — Deserialization Vulnerability Protection in Java, 2021. URL https://medium.com/tech-learnings/serialization-filtering-deserialization-vulnerability-protection-in-java-349c37f6f4

[13] Kantega AS. Notsoserial, 2015. URL https://github.com/kantega/notsoserial.

[14] M. Lessio and A. B. Cve-2017-7525-jackson-deserialization-lab, 2020. URL https://github.com/Ingenuity-Fainting-Goats/CVE-2017-7525-Jackson-Deserialization-Lab.

[15] A. Muñoz and C. Schneider. Serial killer: Silently pwning your java endpoints, 2018.

[16] N. N. I. of Standards and Technology. National vulnerability database nvd, 2022. URL https://nvd.nist.gov/. [Online; accessed 14-February-2022].

[17] OWASP Foundation. Deserialization cheat sheet, 2017. URL https://cheatsheetseries.owasp.org/cheatsheets/Deserialization_Cheat_Sheet.html.

[18] OWASP Foundation. Deserialization of untrusted data, 2017. URL https://owasp.org/www-community/vulnerabilities/Deserialization_of_untrusted_data.

[19] OWASP Foundation. A8:2017-insecure deserialization, 2017. URL https://owasp.org/www-project-top-ten/2017/A8_2017-Insecure_Deserialization.

[20] OWASP Foundation. Owasp top ten web application security risks 2021, 2021. URL https://owasp.org/www-project-top-ten/. [Online; accessed 18-February-2022].

[21] OWASP Foundation. Owasp webgoat, 2021. URL https://owasp.org/www-project-webgoat/.

[22] Precision and recall. Precision and recall — Wikipedia, the free encyclopedia, 2022. URL https://en.wikipedia.org/wiki/Precision_and_recall. [Online; accessed 07-February-2022].

[23] Return To Corporation. Semgrep, 2021. URL https://semgrep.dev/.

[24] A. Sabatini. Benchstress, 2022. URL https://github.com/alex97saba/BenchStress.

[25] C. Schneider. Swat (serial whitelist application trainer), 2016. URL https://github.com/cschneider4711/SWAT.

[26] SonarSource. Sonarlint, 2020. URL https://www.sonarlint.org/.

[27] SonarSource. Sonarqube, 2021. URL https://www.sonarqube.org/.

[28] SonarSource. Sonarsource rules, 2021. URL https://rules.sonarsource.com/java.

[29] M. Woloszyn. Java deserialize webapp, 2016. URL https://github.com/hvqzao/java-deserialize-webapp.

A | Vulnerable application

We present the code of the example of a vulnerable application to deserialization:

Listing 1 Server.java

```java
package hvqzao.java.deserialize.webapp.embedded;

import java.io.File;
import java.util.logging.Level;
import java.util.logging.Logger;

import org.apache.catalina.WebResourceRoot;
import org.apache.catalina.connector.Connector;
import org.apache.catalina.core.StandardContext;
import org.apache.catalina.startup.Tomcat;
import org.apache.catalina.webresources.DirResourceSet;
import org.apache.catalina.webresources.StandardRoot;

public class Server {

	public static void main(String[] args) throws Exception {

		Logger logger = Logger.getLogger("");
		logger.setLevel(Level.WARNING);

		Tomcat tomcat = new Tomcat();
		tomcat.setBaseDir("target/tmp/");
		Connector connector = tomcat.getConnector();
		connector.setProperty("port", "8000");
		//connector.setProperty("address", "127.0.0.1");

		String webappDirLocation = "src/main/resources/webapp/";
		StandardContext ctx = (StandardContext) tomcat.addWebapp("",
				new File(webappDirLocation).getAbsolutePath());

		// Declare an alternative location for your "WEB-INF/classes" dir
		File classes = new File("target/classes");
		WebResourceRoot resources = new StandardRoot(ctx);
		resources.addPreResources(new DirResourceSet(resources,
			"/WEB-INF/classes", classes.getAbsolutePath(), "/"));
		ctx.setResources(resources);

		System.out.println("Running...");
		tomcat.start();
		tomcat.getServer().await();

	}

}
```

Listing 2 Serial.java

```
1  package hvqzao.java.deserialize.webapp.api;
2
3  import java.io.ByteArrayInputStream;
4  import java.io.ByteArrayOutputStream;
5  import java.io.IOException;
6  import java.io.ObjectInputStream;
7  import java.io.ObjectOutputStream;
8  import java.io.Serializable;
9
10 import org.apache.tomcat.util.codec.binary.Base64;
11
12 public class Serial {
13
14         public static Object fromBase64(String s) throws IOException,
15                                                          ClassNotFoundException {
16                 byte[] data = new Base64().decode(s);
17                 ObjectInputStream ois =
18                                 new ObjectInputStream(new ByteArrayInputStream(data));
19                 Object o = ois.readObject();
20                 ois.close();
21                 return o;
22         }
23
24         public static String toBase64(Serializable o) throws IOException {
25                 ByteArrayOutputStream baos = new ByteArrayOutputStream();
26                 ObjectOutputStream oos = new ObjectOutputStream(baos);
27                 oos.writeObject(o);
28                 oos.close();
29                 return new Base64().encodeToString(baos.toByteArray());
30         }
31 }
```

Listing 3 Servlet.java

```
package hvqzao.java.deserialize.webapp.servlet;

import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.ServletOutputStream;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import hvqzao.java.deserialize.webapp.api.Serial;

@WebServlet(name = "MyServlet", urlPatterns = { "" })
public class Servlet extends HttpServlet {

	private static final long serialVersionUID = -1251837460515874243L;

	@Override
	protected void doGet(HttpServletRequest req, HttpServletResponse resp)
					throws ServletException, IOException {
		doPost(req, resp);
	}

	@Override
	protected void doPost(HttpServletRequest req, HttpServletResponse resp)
					throws ServletException, IOException {

		ServletOutputStream out = resp.getOutputStream();
		out.write("<!DOCTYPE html>".getBytes());
		out.write("<html>\n".getBytes());
		out.write("<body>\n".getBytes());
		out.write("<p><a href=\"classpath.jsp\">classpath</a></p>\n"
							.getBytes());
		String data = req.getParameter("data");
		if (data == null) {
			data = Serial.toBase64(new String("text"));
		} else {
			out.write("<p>Deserializing...".getBytes());
			try {
				Serial.fromBase64(data);
			} catch (ClassNotFoundException e) {
				e.printStackTrace();
			}
			out.write("Done!</p>\n".getBytes());
		}
		out.write("<form action=\"/\" method=\"POST\">\n".getBytes());
		out.write(String.valueOf("<p><textarea type=\"text\" name=\"data\">"+
					data+"</textarea></p>\n").getBytes());
		out.write("<p><input type=\"submit\"></p>\n".getBytes());
		out.write("</form>\n".getBytes());
		out.write("</body>\n".getBytes());
		out.write("</html>\n".getBytes());
		out.flush();
		out.close();

	}

}
```

B | BenchStress

We present the code of BenchStress, our web application developed to test the tools:

First is presented the class *Server.java*:

```
1  package insecure.deserialization;
2
3  import java.util.ArrayList;
4  import java.io.IOException;
5  import java.io.ObjectInputStream;
6  import java.io.ObjectOutputStream;
7  import java.lang.ClassNotFoundException;
8  import java.net.ServerSocket;
9  import java.net.Socket;
10 import java.*;
11 import java.lang.System;
12
13 import org.apache.commons.collections4.*;
14 import org.apache.commons.collections4.CollectionUtils;
15
16
17 public class Server {
18
19     public static void main(String args[]) throws IOException, ClassNotFoundException
       ↪  {
20         final boolean val=true;
21         int valInt=6;
22         int i;
23
24         System.setSecurityManager(null);
25
26         ServerSocket server = null;
27         try {
28             server = new ServerSocket(8888);
29         } catch (IOException e) {
30             System.err.println(e.getMessage()); /* port not available*/ return;
```

```
31          }
32
33          System.out.println(server.getInetAddress().getHostAddress());
34          Socket socket = server.accept();
35
36          // Create example notes
37          Note note1 = new Note(1, "Test", "This is the main content");
38          Note note2 = new Note(2, "Shopping list", "Milk\nEggs\nSausage");
39          Note note3 = new Note(3, "Bills", "Don't forget to pay the bills due on the
            ↪  11th!");
40
41          // Create list to store all notes
42          ArrayList<Note> noteList = new ArrayList<Note>();
43          noteList.add(note1);
44          noteList.add(note2);
45          noteList.add(note3);
46          System.out.println();
47
48          // Server loop
49          while (true) {
50              while(true){
51                  // Create streams
52                  ObjectInputStream ois = new
                    ↪  ObjectInputStream(socket.getInputStream());
53                  ObjectOutputStream oos = new
                    ↪  ObjectOutputStream(socket.getOutputStream());
54
55
56                  try (ObjectInputStream ois2 = new
                    ↪  ObjectInputStream(socket.getInputStream())) {     //non trovata
                    ↪  da semgrep
57                      Object o = ois2.readObject();
58                      System.out.println(o.toString());
59                  } catch (Exception e) {
60                      return;
61                  }
62
63                  if (val) {
64                      ObjectInputStream ois3 = new
                        ↪  ObjectInputStream(socket.getInputStream());
65                      Object o3 = ois3.readObject();
66                      System.out.println(o3.toString());
67                  }
68                  else {
```

```
69                      ObjectInputStream ois4 = new
                        ↪  ObjectInputStream(socket.getInputStream());   //viene trovata
                        ↪  da semgrep anche se il codice non arrivera mai qui
70                      Object o4 = ois4.readObject();      //Spotbugs does not find the
                        ↪  vulnerability in the else because val is always true
71                      System.out.println(o4.toString());
72                  }
73
74                  switch (valInt) {
75                      case 1:
76                          System.out.println(valInt);
77                          break;
78                      case 6:
79                          ObjectInputStream ois5 = new
                            ↪  ObjectInputStream(socket.getInputStream());
80                          Object o5 = ois5.readObject();
81                          System.out.println(o5.toString());
82                          break;
83                      default:
84                          break;
85                  }
86
87
88                  // convert object from client to String so we can check it
89                  // not sure if this is safe...
90                  String message=null;
91                  try {
92                      System.out.println(message);
93                  }catch (Exception e){
94                      System.err.println(e);
95                  }finally {
96                      ObjectInputStream ois6 = new
                        ↪  ObjectInputStream(socket.getInputStream());
97                      message = (String) ois6.readObject();
98                      System.out.println(message);
99                  }
100
101                 ObjectInputStream ois6 = (val) ? new
                    ↪  ObjectInputStream(socket.getInputStream()): null;
102                 message = (!val) ? null: (String)ois6.readObject();
103                 System.out.println(message);
104
105                 ObjectInputStream[] oisArray = new ObjectInputStream[10];
106                 //oisArray[0] = new ObjectInputStream(socket.getInputStream());
                    ↪  found from semgrep
```

```
107                 for (i=0; i<10; oisArray[i] = new
                    ↪  ObjectInputStream(socket.getInputStream())) {     //not found
                    ↪  from semgrep
108                     Object o= oisArray[i].readObject();
109                     i++;
110                 }
111
112                 do {
113                     ObjectInputStream ois8 = new
                        ↪  ObjectInputStream(socket.getInputStream());
114                     message = (String) ois8.readObject();
115                     i+=10;
116                 }while (i<100);
117
118                 /*
119                 ByteArrayOutputStream bOutput = new ByteArrayOutputStream(12);
120                 byte[] b = bOutput.toByteArray();
121                 ByteArrayInputStream bytesIn = new ByteArrayInputStream(b);
122                 ObjectInputStream objIn = new ObjectInputStream(bytesIn);
123                 Object obj = objIn.readObject();
124                 System.out.println(obj.toString());
125                  */
126
127                 if(message.equalsIgnoreCase("GET")){
128                     // Client wants all of the saved notes
129                     oos.writeObject(noteList);
130                     continue;
131
132                 }else if (message.equalsIgnoreCase("SAVE")){
133                     // Client wants to save a note, accept a Note object
134                     Note newNote = (Note) ois.readObject();
135                     noteList.add(newNote);
136                     continue;
137
138                 }else if (message.equalsIgnoreCase("BYE")){
139                     break;
140                 }else{
141                     // Ignore
142                 }
143             }
144         }
145     }
146 }
```

Then, we show the *Note.java* class:

```java
package insecure.deserialization;

import java.io.Serializable;

public class Note implements Serializable{
    private Integer note_id;
    private String note_title;
    private String note_body;

    // Constructor
    public Note(Integer id, String title, String body){
        note_id = id;
        note_title = title;
        note_body = body;
    }

    public void print_note(){
        System.out.println(note_title + "\n" + note_body + "\n");
    }
}
```

Finally, it is presented the *Client.java* class:

```java
package insecure.deserialization;


import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.Socket;
import java.net.UnknownHostException;
import java.util.ArrayList;
import java.util.Scanner;


public class Client {
    public static void main(String[] args) throws UnknownHostException, IOException,
    ↪ ClassNotFoundException, InterruptedException{
        //get the localhost IP address, if server is running on some other IP, you
        ↪ need to use that
        Socket socket;
```

```
17          ObjectOutputStream oos;
18          ObjectInputStream ois;
19
20          // Args
21          String server_ip = null;
22          int port = 0;
23
24          // If you want to launch it locally
25          server_ip = "127.0.0.1";
26          port = 8888;
27
28          /* If you want to choose server_ip and port
29          if (args.length==2) {
30              try {
31                  port = Integer.parseInt(args[1]);
32              } catch (NumberFormatException e) {
33                  System.err.println("Argument" + args[1] + " must be an integer.");
34                  System.exit(1);
35              }
36              server_ip = args[0];
37          } else {
38              System.err.println("Usage: client <server_ip> <port>");
39              System.exit(1);
40          }
41          */
42
43          // Scanner
44          Scanner scan = new Scanner(System.in);
45          // Setup socket
46          System.out.println("Connecting to server @ " + server_ip + ":" + port +
            ↪  "...");
47          socket = new Socket(server_ip, port);
48          System.out.println("Connected!\n");
49          System.out.println("WELCOME TO NOTE KEEPER CLIENT V1.42\n");
50
51          while (true){
52              System.out.println("========= ACTIONS =========");
53              System.out.println("1    - Print all notes");
54              System.out.println("2    - Add a new note");
55              System.out.println("EXIT - Exit this program\n");
56              System.out.print("Select an action to perform: ");
57              String selection = scan.nextLine();
58
59              // Setup streams
```

```
60              oos = new ObjectOutputStream(socket.getOutputStream());
61              ois = new ObjectInputStream(socket.getInputStream());
62
63              // Do action
64              if (selection.equals("1")){
65                  // Print all notes from server
66                  // Tell server we want all the all the notes
67                  oos.writeObject("GET");
68                  oos.flush();
69                  Thread.sleep(100);
70                  // Save the data from the server
71                  ArrayList<Note> noteList = (ArrayList<Note>) ois.readObject();
72                  // Print all the notes
73                  System.out.println("========================================");
74                  System.out.println("================ NOTES ================");
75                  System.out.println("========================================\n");
76
77                  for (Note n : noteList){
78                      n.print_note();
79                  }
80
81                  System.out.println("\n========================================");
82                  System.out.println("============== END NOTES ==============");
83                  System.out.println("========================================\n");
84                  continue;
85              } else if ( selection.equals("2")){
86                  // Send a new note to the server
87                  System.out.println("Please enter the following information to create
                    ↪  a new note...");
88                  System.out.print("Title: ");
89                  String newNoteTitle = scan.nextLine();
90                  System.out.print("Body: ");
91                  String newNoteBody = scan.nextLine();
92                  Note newNote = new Note(42, newNoteTitle, newNoteBody);
93                  oos.writeObject("SAVE");
94                  Thread.sleep(100);
95                  oos.writeObject(newNote);
96                  System.out.println("The note has been sent to the server!");
97              } else if (selection.equalsIgnoreCase("exit")){
98                  // Quit
99                  System.out.println("Exiting...");
100                 break;
101             } else {
102                 // Invalid action
```

```
                System.out.println("Invalid Action...");
                continue;
            }
        }
    }
}
```