Java Serialization

Chapter · May 2022

DOI: 10.1007/978-3-030-98467-0_7

1 author:

Dirk Pawlaszczyk
Hochschule Mittweida
36 PUBLICATIONS **147** CITATIONS

SEE PROFILE

Chapter 7
Java Serialization

Dirk Pawlaszczyk

Abstract Java Serialization is a popular technique for storing object states in the Java programming language. In the field of mobile forensics, we come across such artefacts. App developers very often resort to this technique to make their application state persistent. Serialization is also used when transferring data over a network between two Java applications using Remote Method Invocation(RMI). In the past, there have been recurring security issues associated with this technology. Despite its importance for forensic casework, one can hardly find any literature on this topic. In this chapter, we give an insight into the binary format. For this purpose, special features of the format are presented using an example. In addition to the actual protocol structure, basic steps for acquiring such data and analyzing it will be discussed. Practical hints for searching serials are given. Finally, the security issues are addressed.

7.1 Introduction

Among app developers, the Java programming language has been the first choice for many years. The popularity of the language can be attributed to its simple syntax and compelling framework. As with any object-oriented language, the execution state of the program is managed through objects. From time to time, an application needs to back up its current state to disk. Of course, it is possible to store important data in a database such as SQLite. However, this usually requires object-relational mapping to be performed first. From the beginning, Java offers an alternative for persistent writing of objects: the so-called Java Object Serialization (JOS) [93]. Java's standard serialization seems to be a good choice, especially for app developers who want to store objects' current execution state. By serialization, we understand the ability to

Faculty for Applied Computer Science
University of Applied Sciences (Hochschule Mittweida), Technikumplatz 17, 09648 Mittweida, Germany, e-mail: pawlaszc@hs-mittweida.de

C. Hummert, D. Pawlaszczyk (eds.), *Mobile Forensics – The File Format Handbook*,
https://doi.org/10.1007/978-3-030-98467-0_7

convert an object in the application's main memory into a format that allows the object to be written to a file or transported over a network connection.

Since many apps rely on this format by default to store their program data, investigators are of particular interest. This chapter will take a closer look behind the scenes at the Standard Serialization concept in Java. We pay special attention to the binary format and explore how to analyze this file type.

7.2 Object Serialization in Java

7.2.1 Serialization Techniques in Java

Under Java SE, objects can be automatically mapped and stored persistently using various approaches [91], [54]:

- **Standard serialization:** The object structure and states are saved in a binary format. As already mentioned, this procedure is also called Java Object Serialization (JOS). Standard serialization is very important for remote method calls and storing things over time and then retrieving them from the closet at some point.
- **XML serialization via JavaBeans Persistence:** JavaBeans - and only such - can be saved in an XML format. One solution is JavaBeans Persistence (JBP), which was originally intended for Swing. When the state of a graphical user interface is binary persisted with JOS, changes to the Swing API's internals are not easily possible since the binary format of JOS is very tightly coupled with the object model. That is, objects sometimes cannot be reconstructed from the binary document. JBP decouples this by communicating only through setters/getters and not on internal references, which are an implementation detail, which can change at any time. Nowadays, JBP hardly plays a role in practice.
- **XML mapping via JAXB:** With JAXB, a second API is available for mapping the object structure to XML documents. The eXtensible Markup Language (XML) supports a text-based data format based on markups. The platform-independent exchange format is part of the standard library from version 6. It is a fundamental technology, especially for Web service calls.

All three options are already built into Java by default. The standard object serialization creates a binary format and is very strongly oriented towards Java. Other systems cannot do much with the data. XML is convenient as a format because other systems can process it. Another compact binary format that also allows interoperability is Protocol Buffers [1] from Google. The company uses it internally when different applications are to exchange data.

Finally, objects can also be stored in relational databases called object-relational mapping (OR mapping). This technique is very sophisticated because the object

[1] http://code.google.com/p/protobuf/

models and tables are quite different. The Java SE does not offer any support for OR mapping, but it can be done with additional frameworks, such as the JPA (Java Persistence API).

7.2.2 Serialization by Example

The traditional way from an object to persistent storage is via Java's serialization mechanism [57][54]. JOS is the technology we want to deal with in the following. The standard serialization offers a simple possibility to make objects persistent and to reconstruct them later. The object state (no static ones!) is written into a byte stream (serialization). From this, it can be reconstructed to an object again later (deserialization). The object state is written into a serial data stream of 0 and 1. Java provides two special classes for this purpose: *ObjectOutputStream* and *ObjectInputStream* with a *writeObject()* respectively *readObject()*-method. Both classes can be found in the *java.io* package of the Java standard class library [2]. To save an object's state, we must pass the object reference as a parameter to the writeObject()-method. In the Java ecosystem, the applications programmers are encouraged to use serialization almost everywhere.

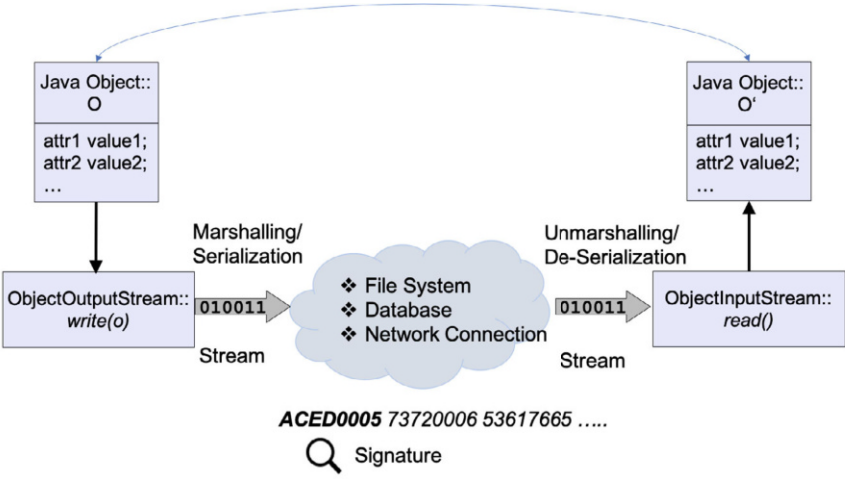

Fig. 7.1: Java Object Serialization (JOS) - Concept

Serialization concept can show its strengths, especially in communication between different Java processes distributed over a network. We serialize some of the objects, send them to another process for processing, serialize the transformed object and send

[2] https://docs.oracle.com/javase/7/docs/api/java/io/ObjectOutputStream.html

it back. To illustrate this, we will discuss a small example program. In the following, we want to make a class *SaveMe* serializable. For this, we need the following code:

Listing 7.1: Class definition of a class to serialize

```
import java.io.Serializable;

public class SaveMe implements Serializable {

        private static final long serialVersionUID = 1L;
        private int x;
        private double d;
        private String s;

        public SaveMe() {
                this(100, 3.14, "hello");
        }

        public SaveMe(int x, double d, String s) {
                this.x = x;
                this.d = d;
                this.s = s;
        }

        public String toString() {
                return s + " " + x + " " + d;
    }
}
```

In Listing 1.1, the class *SaveMe* is defined first. In order for objects to be Serialized, the classes must implement the *Serializable* interface. This interface thus serves as a marker to indicate that the class can be Serialized.

! Attention

When serializing an object, only its attributes are stored. Methods or program code remains in the *.class* file.

Java is assigning a serial number to each object of the class it writes to a stream. This serial number is then used to re-create the class when it is reread. If two variables contain references to the same object and we write the objects to a file and later read them from the file, then the two objects that are read will again be references to the same object. All attributes of an object can be made persistent in this way. However, there are two exceptions. Attributes defined with the prefixed key *transient* are not Serialized. This identifier was explicitly introduced to exclude an attribute from Serialization. It can be helpful, for example, if confidential or volatile data should not be saved. The second exception is class attributes preceded by the keyword *static*. With one exception, such attributes shared by all objects of a class are not Serialized. In our example, there are no transient attributes and only one static class attribute. Since the serialVersionUID property is defined as

static and thus should not be stored. In this case, however, an exception is made. In Java, serialVersionUID is like version control, ensuring that both Serialized and deSerialized objects use the compatible class. For example, if an object is saved into a stream with serialVersionUID=1L, when we convert the stream back to an object, we must use the same serialVersionUID=1L. Otherwise, an InvalidClassException is thrown.

If we create a *SaveMe* object $o1$ and call $writeObject(o1)$, the ObjectOutputStream pushes the variable assignments (here x, d and s) into the data stream. An example is shown in the next listing:

Listing 7.2: Output Class 'Serializer'

```
import java.io.*;

public class Serializer {

public static void main(String[] args) throws IOExectpion {
                ObjectOutputStream o =
                    new ObjectOutputStream(
                            new FileOutputStream("saved.ser"));
                SaveMe o1 = new SaveMe();
                o.writeObject(o1);
                o.close();
}}
```

This routine creates a file, *saved.bin*, on the disk that contains the serialized object. With a few lines, the state of our object $o1$ can be saved to a file. In the example shown, the serialized object is written to a file. To send the object over the network, we have to create an object of the Socket class and start writing to the output stream of this class:

Listing 7.3: Output to a socket connection instead of a file

```
...
Socket connection = new Socket(hostName, portNumber);
OjbectOutputStream oos =  new
    ObjectOutputStream(connection.getOutputStream(), true);
oos.writeObject(o1);
```

We only need to adjust two lines in our program, and we are ready to go. It could not be simpler. However, this form of serialization also has disadvantages. Standard serialization works according to the principle: Everything reached from the object graph enters the data stream serialized. Suppose the object graph is extensive, the time for serialization and the data volume increase. Unlike other persistence concepts, it is not possible to write only the changes. For example, if only one attribute value has changed in an extended object list, the entire list must be rewritten. This is not efficient. However, let us focus on analyzing the binary format.

The output file <saved.ser> from the above example has a size of 80 bytes. The content of the file can be seen in Fig. 7.2. In addition to the actual attribute values, information about data type and class type is also stored in the file. Fig. 7.3 offers a high-level look at the serialization algorithm for this example. In the next section,

```
00  AC ED 00 05 73 72 00 06 53 61 76 65 4D 65 00 00 00   ....sr..SaveMe...
11  00 00 00 00 01 02 00 03 44 00 01 64 49 00 01 78 4C   ........D..dI..xL
22  00 01 73 74 00 12 4C 6A 61 76 61 2F 6C 61 6E 67 2F   ..st..Ljava/lang/
33  53 74 72 69 6E 67 3B 78 70 40 09 1E B8 51 EB 85 1F   String;xp@...Q...
44  00 00 00 64 74 00 05 68 65 6C 6C 6F                  ...dt..hello
```

Fig. 7.2: Hex view of the serialized object $o1$

we will take a closer look a the serialized format of the object and see what each byte represents.

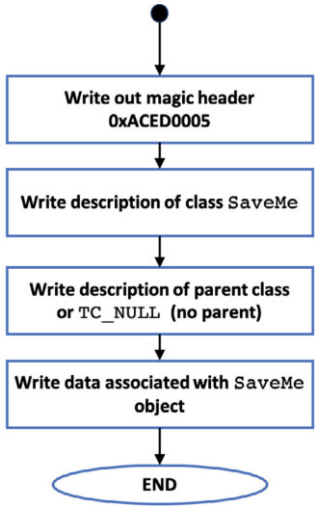

Fig. 7.3: Outline of the Serialization steps for class *SaveMe*

7.3 Java Object Serialization Protocol Revealed

As already discussed, Java's object Serialization creates a binary stream. Unlike JavaBeans persistence, for example, it is not readable by humans. Fortunately, the format is well documented. Oracle provides corresponding documentation on its website in which details of the *Object Serialization Stream Protocol* are presented [56]. The specification defines context-free grammar for the stream format. It gives a good insight into the Serialization process. The stream rules formulated in it are used directly in the Serialization of an object. In addition, a look at the source code of the *ObjectOutputStream* class reveals a lot about concrete implementation. Fig. 7.4 shows the first part of the grammar using a syntax diagram. It defines a set of

production rules (<R_n>). These rules can then be used directly to generate an object data stream.

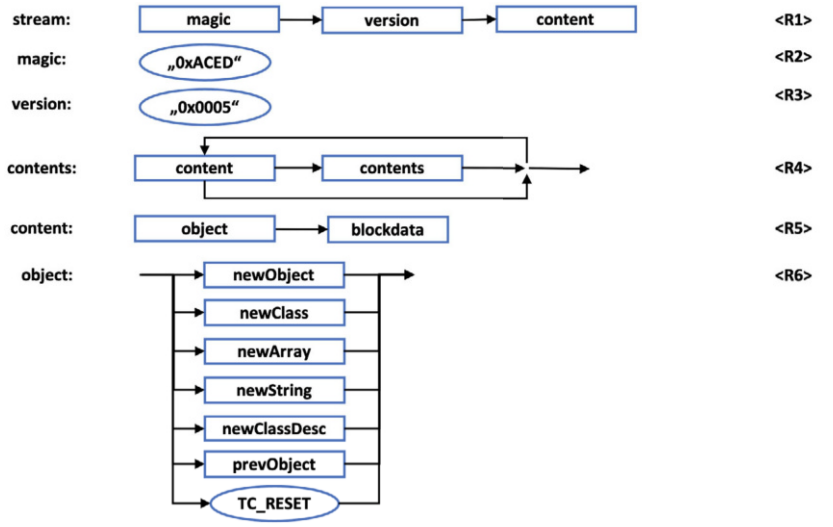

Fig. 7.4: Syntax-diagram for the Java Object Serialization Protocol (Header Detail)

The syntax diagram for this is thus as follows: Definitions of symbols are followed by a ":". We first distinguish between the terminal and non-terminal symbols. The latter can be recognised by the fact that they consist of a literal enclosed in double quotes. Constant values are enclosed in an oval. A rectangle marks non-terminals. A sequence of values is represented as a series of symbols on the same line. The individual values can be found precisely in this order in the stream. A definition consists of zero or more alternative values. Alternatives are indicated by a branch (exclusive OR).

Let us now turn to the concrete meaning of the rules. Each object stream initially consists of a magic number, the version number and the actual content specification (R_1). A magic number opens the data stream. The 2-byte integer value resides on offset 0. On offset two, the stream version field follows. According to the internal specification, this is assigned to value *0x0005* (R_3). Thus, a Java object stream can be detected with the help of the header signature 0XACED0005 (R_2 + R_3). This information is beneficial when carving to serial format files on disk. The actual content directly follows the header in the data stream.

The *contents* field is defined recursively (see R_4). Thus it can hold multiple content objects. A *content* object is first divided into an object description and a data block with the concrete attribute values (R_5). Thus, valid values for a content element are *objects, classes, arrays, strings, enumerations, exceptions* (R_6). In this way, all elements of a class and its objects can be described. Within the byte stream,

limiter symbols indicate the type, start and end of particular elements. These terminal constants (TC) are shown in Table 7.1.

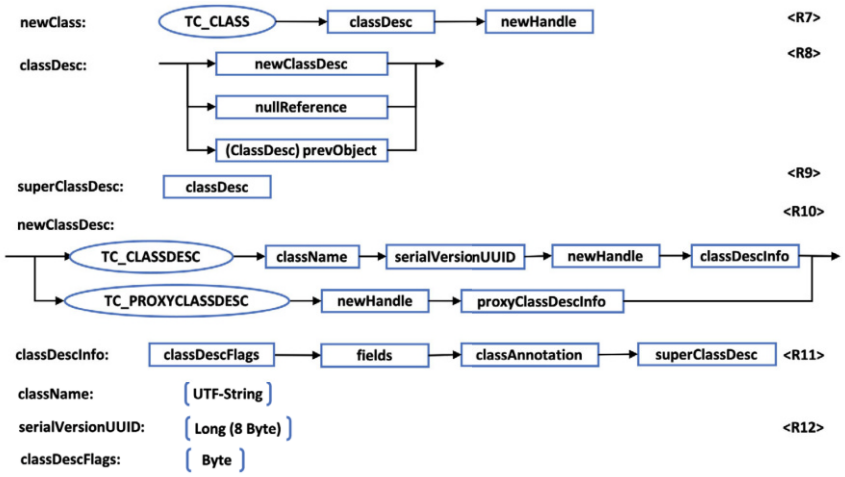

Fig. 7.5: Syntax-diagram for the 'newClass'production rule

Table 7.1: Stream Terminal Constants (TC)

Constant	Value(hex)
TC_NULL	0x70
TC_REFERENCE	0x71
TC_CLASSDESC	0x72
TC_OBJECT	0x73
TC_STRING	0x74
TC_ARRAY	0x75
TC_CLASS	0x76
TC_BLOCKDATA	0x77
TC_ENDBLOCKDATA	0x78
TC_RESET	0x79
TC_BLOCKDATALONG	0x7A
TC_EXCEPTION	0x7B
TC_LONGSTRING	0x7C
TC_PROXYCLASSDESC	0x7D
TC_ENUM	0x7E
baseWireHandle	0x7E0000

Fig. 7.5 describes the syntax definition for a class. Besides the class name, the *SerializationUUID* and *ClassDescInfo* elements are of particular importance (see R_{10}). Note: A corresponding class description must first be placed in the data stream

for each object to be Serialized. It contains information about attribute names and data types. If the class has been derived from a special super-class, the class must, of course, also be Serialized. Since attributes of the super-class are also inherited in the deriving class, we must also capture them. If the super-class, in turn, has a parent class, then this must also be described. Inheritance relationships thus significantly increase the data stream. Fortunately, Java supports only single inheritance.

Particular attention should be paid to the symbol TC_CLASSDEC. It is used to show the start of a new class definition. The byte TC_OBJECT (0x73) represents the start of an object. A data block (TC_BLOCKDATA) is in turn initiated by the byte value 0x77. We need to search the serial stream for these symbols to make the data visible. The constant *baseWireHandle* is of particular importance. Each Serialized element is assigned to such a handle. The first Serialized element contains the handle 0x7E0000, the next object is the trade 0x7E00001 and so on. In this way, for example, an object can reference its class.

Table 7.2: Type Codes / Stream Symbols for Primitive Types

Symbol	Datatype
B	byte
C	char
D	double
F	float
I	integer
J	long
S	short
Z	boolean

Table 7.2 shows the identifiers used in the stream for the eight built-in data types in Java. We can now start decoding the first part of our sample file <saved.ser> with what we have discussed so far (see below). As discussed above, the binary stream is opened by the magic number (0xACED) and the stream version (0x0005). The next byte indicates that this is an object that follows (0x73). The following byte introduces the class identifier (0x72).

> Important

Java serialized objects have a specific signature. We can use it to identify an object stream. The binary value is 0xACED0005. It translates to BASE64 as “rO0ABQ==” in a HTTP-Stream for example.

The sub-element is composed of the className. The class name is again composed of a length specification 0x0006 and the actual name string 0x536176654D65.

```
   |0x00|ACED0005 73720006 53617665 4D650000 |....sr..SaveMe..|
   |0x10|00000000 00010200 03440001 64490001 |.........D..dI..|
   |0x20|784C0001 73740012 4C6A6176 612F6C61 |xL..st..Ljava/la|
   |0x30|6E672F53 7472696E 673B7870 40091EB8 |ng/String;xp@...|
   |0x40|51EB851F 00000064 74000568 656C6C6F |Q......dt..hello|

STREAM_MAGIC - 0xACED
STREAM_VERSION - 0x0005
Contents
  TC_OBJECT - 0x73
    TC_CLASSDESC - 0x72
      className
        Length - 6 - 0x0006
        Value - SaveMe - 0x536176654d65
      serialVersionUID - 0x0000000000000001
      newHandle 0x007E0000
      classDescFlags - 0x02 - SC_SERIALIZABLE
      fieldCount - 3 - 0x0003
      Fields
        0:  Double - D - 0x44
            field name 'x'
            Length - 1 - 0x00 01
            Value - d - 0x64
        1:  Int - I - 0x49
            field name 'd'
            Length - 1 - 0x00 01
            Value - x - 0x78
        2:  Object - L - 0x4C
            field name 's'
            Length - 1 - 0x00 01
            Value - s - 0x73
            class name
            TC_STRING - 0x74
              newHandle 0x007E0001
              Length - 18 - 0x00 12
              Value - Ljava/lang/String; -
              0x4C6A6176612F6C616E672F537472696E673B
      classAnnotations
        TC_ENDBLOCKDATA - 0x78
      superClassDesc
        TC_NULL - 0x70. <- end of class description
```

integer. The value in the example is 0x0000000000000001. The *SaveMe* class is internally assigned with the handle 0x007e0000. The handle never appears directly in the stream. Only if later, a stream element again refers to the class will be visible in the stream. Various flags typically follow the class name. This flag indicates that this class supports serialization (0x02). Now we have to read out the actual attribute values. They follow directly after the class description. The object we stored in the above example has a total of three more attributes: 'x', 'd' and 's'. Since the serial number attribute is not counted, the property *fieldcount* has the value 0x0003. The individual attribute descriptions with a data type, name and length specification

follow directly afterwards (see Fig. 7.6). The string attribute is different here. Since this value itself is an object and not a primitive data type, it is also handled. Again, this is not displayed in binary code. However, Java carries an internal counter. The TC_ENDBLOCKDATA (0x78) value marks the end of the class description. Next, the serialization algorithm checks to see if the current class has any parent classes. If it did, the algorithm would start writing that class. Since we have not specified a superclass, the superClassDesc field remains empty or is assigned the terminal symbol TC_NULL(0x70). Finally, the actual attribute values for our object *o1* are still missing:

```
|0x30|6E672F53 7472696E 673B7870 40091EB8 |ng/String;xp@...|
|0x40|51EB851F 00000064 74000568 656C6C6F |Q......dt..hello|

 newHandle 0x00 7e 00 02
    classdata
      SaveMe
        values
          'd' (double)3.14 - 0x40091EB851EB851F
          'x' (int)100 - 0x00000064
          's' (object) TC_STRING - 0x74
                newHandle 0x007E0003
                Length - 5 - 0x0005
                Value - hello - 0x68656C6C6F
```

As we surely noticed, the object name *o*1 is missing. This information is not stored in the stream since it is only an identifier. The programmer decides under which identifier the object can be accessed after deserialization. The value assignment of the ordinal types *d* and *x* follows directly in the stream. The first attribute has 8 bytes in length for the LONG value. The integer value has a length of 4 bytes. Thus, unlike database formats such as SQLite, no compression is used. The memory content is transferred 1:1 into the stream. The string value terminates the stream. A string is not an ordinal type but an object itself. Therefore first, the identifier follows as a string (0x74). The length of a string is dynamic. Therefore the length specification is additionally prefixed to the string value (0x05). The actual value follows last. There is no particular end identifier. Instead, the stream ends [54].

There are some production rules which are not listed so far. Special stream types like enumerations, exceptions or proxy classes are missing. At this point we refer to the protocol description on the oracle website [57][56].

7.4 Pitfalls and Security Issues

In the last years, the serialization protocol of Java was increasingly in the criticism due to different vulnerabilities [76][34]. At this point, we want to shed light on the background and show how these threats can be minimized.

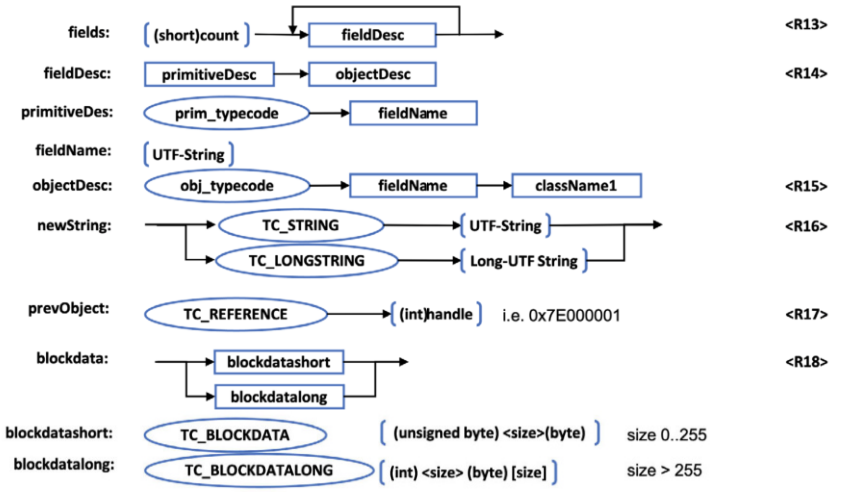

Fig. 7.6: Syntax-diagram for the objects 'fields'and 'blockdata'

7.4.1 Hands on Serialized Objects

With what we already know about serialization, it is easy to find a point of attack. It is not uncommon, although it is not recommended, to transmit or store important confidential information in streams. Assume a user-supplied serialized object we discovered. We can manipulate program logic by tampering with the information stored within the stream. For example, let's take our object *SaveMe*. We can easily modify the string at the end of the stream:

```
0005 68656C6F | ..hello    ->  0007 62656576 696C  | ..beevil
```

Since there are no parity bits or checksums, we do not need to adjust anything else within the stream. In our case, that is probably not a big deal. If somebody cheats on a gaming app and overwrite the high score, we can certainly get over it. However, the whole thing changes quickly when confidential data is included in our stream (e.g. user=admin, password=abc123). For example, if the Java object is used as a cookie for access control, we can change the usernames, role names, and ID-token. One can also try tampering with any value in the object that is a file path. We can even alter the program's flow if we override the correct field.

7.4.2 Beware of Gadget Chains

As if that were not enough, we can sometimes even perform remote code execution (RCE) [34][76]. In Java applications, so-called *gadget classes* can be found in the libraries loaded by the application. Using gadgets that are in-scope of the application,

we can create a chain of method invocations that eventually lead to RCE. This chain can be bumped during or after the deserialization process.

Listing 7.4: Possible Vulnerable Class

```
public class Vulnerable{
    public Object invoke(SaveMe o) {
        return Runtime.exec("echo \"I just want to say\"" + o.s);
    }
}
```

An example class is shown in the listing above. The *invoke()* method in this example uses the string attribute we modified earlier. If we set the string *s* to "hello | mkdir somedirectory", the second part of the statement causes the creation of a new directory on the target system. This sort of attack is called *Command injection*. Alternatively, we can, of course, execute any command we like. The goal is to execute arbitrary commands on the host operating system via a vulnerable process. Web servers are very prone to this form of attack.

All we need to find is an appropriate hooking point. Therefore, we should look for gadgets in commonly available libraries to maximize the chances that this gadget is in-scope of the application. To date, exploits utilizing gadgets are already known and published. Those classes are mostly part of popular libraries such as the Apache Commons-Fileupload, Apache Commons-Collections, or Groovy. A collection with the gadget chains for Java can be found in the *ysoserial* project from Chris Frohoff [3]. The repository offers a collection of utilities and gadget chains discovered in shared Java libraries. Due to unsafe serialization, a gadget chain may automatically be invoked and cause the command to be executed on the host system. The creation of an unsafe serial object with *ysoserial* is straightforward:

```
$ java -jar ysoserial.jar [gadget chain] '[command to execute]'
```

The dangerous thing about this is that it does not depend on what classes we use in our application. It is sufficient that the class in question is accessible via the local classpath. To honour the rescue of Java developers, however, it must be said that this is not only a particular problem of Java Runtime Environment. Such security issues can also be found in languages like Python, PHP, or Ruby [92],[87].

However, how can we prevent such attacks now? One measure is to blocklist or allowlist object classes before deserializing them. Most suitable for this is the *resolve()* method of the ObjectInputStream class (see Fig. 7.7). If we would validate the object directly after *readObject()* has finished its work, it may already be too late. However, if serialization is performed by a framework class working in the background, we do not even have to notice it.

[3] https://github.com/frohoff/ysoserial

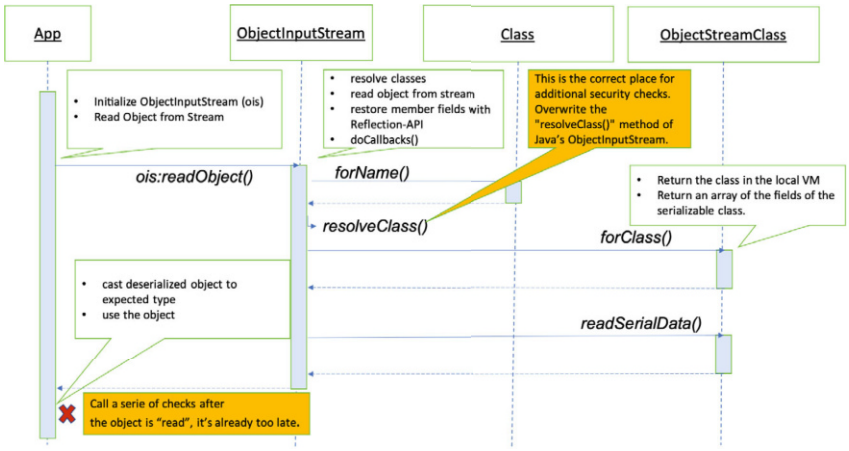

Fig. 7.7: Sequence Diagram of Object De-Serialization

7.5 Conclusions

In this chapter, insight into the standard Serialization format of Java was given. Serialized objects are used in many places in Java. Despite the security problems mentioned earlier and the relatively modest performance, the format enjoys unbroken popularity. Forensically, the file format is exciting because it is not uncommon for confidential or sensitive data to be stored in a stream. However, it should not be a problem to restore attribute assignments with the appropriate tools, even for unknown classes.