École Polytechnique Fédérale de Lausanne

Java Deserialization Fuzzing

by Bastien Wermeille

Master Project Report

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer
Thesis Advisor

Prashast Srivastava
Thesis Supervisor

EPFL IC IINFCOM HEXHIVE
BC 160 (Bâtiment BC)
Station 14
CH-1015 Lausanne

January 7, 2022

Acknowledgments

I greatly thank my PhD supervisor Prashast Srivastava for his great help, support and good advice during this project. Furthermore, I would like to thank professor Mathias Payer for having proposed this project and for providing this nice thesis template.

Lausanne, January 7, 2022 Bastien Wermeille

Abstract

Deserialization of unsecured data has one of the highest weighted impacts from Common Vulnerability and Exposures. However, there currently exists no automated way to identify with high precision such vulnerabilities. The goal of this project is to challenge 3 key buildings blocks to later build an automated fuzzer allowing to detect such exploits. The first steps consist of the analysis of existing gadget chains to identify patterns that will ease the creation of a sanitizer for detecting exploits. The second step was the creation of a sanitizer that was tested on existing gadget chains by using the result of the analysis that identified 2 main sink gadgets that were used in 90% of the analysed chains. Finally, the last part focused on the extensions of jazzer to ease the generation of custom fake objects for the fuzzer that will be used to detect exploits.

Contents

Chapter 1

Introduction

Serialization is the process of transforming a local object into a sequence of bytes. This allows saving the state of the object to store or transmit it. The reverse operation is called deserialization and might be conducted on another computer with a different OS.

Insecure deserialization is part of the top ten OWASP 2021 in the category Software and Data Integrity failure[11][10], this type of vulnerability happen when some untrusted data is deserialized and lead to some unwanted behaviour, like arbitrary code execution.

CVE-2015-8103 is a vulnerability identified in the Jenkins CLI subsystems allowing arbitrary remote code execution using special crafted serialized object[1][8]. This vulnerability was caused by the insecure deserialization of untrusted objects. This type of vulnerability has been identified for the first time in 2006[12], but it is only in 2015 that the interest has grown for it due to the work *Marshalling Pickles* at AppSecCali2015[3]. Since then, this topic has been discussed at security conferences[6] and research has been conducted[4].

For the specific case of Java, the deserialization process triggers some magic methods behind the scene that can be exploited to gain arbitrary code execution. This topic is critical because vulnerabilities have a high impact and there is no real solution to prevent this except to stop using serialization.

The idea of this project is to develop an automated way of identifying those types of vulnerabilities in the form of a fuzzer. The project will be centered on the language Java. To develop such a tool, we identified the following basic tasks:

- Categorisation of Gadgets
- Sanitizer
- Fuzzer

- Pattern detection
- Custom fake object generation

In this project, I will focus on the first categorisation of gadgets to identify what to look for during the exploits and identify the most used gadgets by analysing known gadget chains. Then once the main gadgets are identified, I will develop a sanitizer, to detect such exploits. Finally, the last part on which I will work is the custom fake object generation. The idea is to provide a way of generating some custom fake objects that will be provided to the deserialization process. Those objects will trigger the exploitation and be detected by the sanitizer previously developed.

Chapter 2

Background

of Ian Haken in the paper Automated Discovery of Deserialization Gadget Chains.

Java introduced the serialization in JDK version 1.1, which allow making an object or a collection of objects persistent for storage or network exchange. The serialized object can then be deserialized later in another *Java Virtual Machine* (JVM) independently of the *Operating System* (OS). This functionality is used with *Remote Method Invocation* (RMI).

With JAVA, an object is serializable if it implements the interface *java.io.Serializable* or *java.io.Externalizable*, no additional constraint is required. Now let's imagine the simple case where we have a web server serving requests :

```
@POST
public String renderUser(HttpServletRequest request) {
  ObjectInputStream ois = new ObjectInputStream(
    request.getInputStream());
  User user = (User) ois.readObject();
  return user.render();
}

public class User {
  private String name;
  public String render() {
    return name;
  }
}

public class ThumbnailUser extends User {
  private File thumbnail;
  public String render() {
    return Files.read(thumbnail);
  }
}
```

Figure 2.1 – Deserialization vulnerability example[4]

We have 2 classes, *User* and *ThumbnailUser*. The issue here is that we have full control over the data that is deserialized. This means that it is possible to change the variable *thumbnail* in the serialized form to any filename we prefer. As a result, it is possible to have access to every file the JVM has right on. This exploit does not rely on any gadgets chains but emphasises that having control over the Data Types means controlling the code.

The main security issue with deserialization is not this specific case but rather the fact that during deserialization itself, some magic methods are invoked. This includes *readObject* or *readResolve* but many serializable JDK classes that implement those methods, trigger methods on the deserialized object themself. For example, the deserialization of a *HashMap* will trigger the following methods on the object it contains:

- Object.hashCode()
- Object.equals()

Or for a *PriorityQueue* :

- Comparator.compare()
- Comparator.compareTo()

All those objects provide additional entry points for potential gadgets chains. Here is a real example of gadget chain identified in Clojure :

```
public class HashMap<K,V> implements Map<K,V> {
  private void readObject(ObjectInputStream s) {
    int mappings = s.readInt();
    for (int i = 0; i < mappings; i++) {
      K key = (K) s.readObject();
      V value = (V) s.readObject();
      putVal(key.hashCode(), key, value);
    }
  }
}
```

```
public class AbstractTableModel$ff19274a {
  private IPersistentMap __clojureFnMap;
  public int hashCode() {
    IFn f = __clojureFnMap.get("hashCode");
    return (int)(f.invoke(this));
  }
}
```

```
public class FnConstant implements IFn {
  private Object value;
  public Object invoke(Object arg) {
    return value;
  }
}
```

```
public class FnCompose implements IFn {
  private IFn f1, f2;
  public Object invoke(Object arg) {
    return f2.invoke(f1.invoke(arg));
  }
}
```

```
public class FnEval implements IFn {
  public Object invoke(Object arg) {
    return Runtime.exec(arg);
  }
}
```

Figure 2.2 – Gadget chain identified in Clojure[4]

In this scenario, the deserialization of a *HashMap* will trigger *hashCode* on an *AbstractTableModel$ff19274a*. This method will then invoke an *FnEval* object using the result returned by the invocation of an *FnConstant* object. This *FnEval* will then trigger *Runtime.exec* with the provided parameter previously returned. As all those objects are deserialized, we have full control over those, meaning that we can change the variable value of the *FnConstant* object, allowing us to have remote code execution.

This example is pretty straightforward but can be much more complex by exploiting gadgets from multiple independent libraries. Therefore, the more packages, the more potential gadget chains. For example, in a presentation made by Ian Anken from Netflix, he presented a gadget chain spanning over 9 independent libraries allowing random code execution[4].

Note: the examples presented here come from the work of Ian Haken[4].

Chapter 3

Gadget Categorization

In this section, I will present the gadget analysis results. I analysed existing gadgets chains in 2 steps. During the first one, I mainly focussed on identifying the *source gadget* (first gadgets of the chain) and the *sink gadget* (last gadget allowing the exploit) by using the known chains from the Ysoserial collection[2] and reproducing the exploit.

Gadget chain	Source	Sink
CommonsBeanutils1	PriorityQueue	TemplatesImpl
Clojure	HashMap	clojure.main$eval_opt
Groovy1	PriorityQueue	Runtime.exec
CommonsCollections1	AnnotationInvocationHandler	Runtime.exec
CommonsCollections2	PriorityQueue	Runtime.exec
CommonsCollections3	AnnotationInvocationHandler	Runtime.exec
CommonsCollections4	PriorityQueue	Runtime.exec
CommonsCollections5	BadAttributeValueExpException	Runtime.exec
CommonsCollections6	HashSet	Runtime.exec
CommonsCollections7	HashTable	Runtime.exec
Hibernate1	HashMap	GetterMethodImpl
Hibernate2	HashMap	GetterMethodImpl
JBossInterceptors1	InterceptorMethodHandler	TemplatesImpl
JSON1	HashMap	TemplatesImpl
JavassistWeld1	InterceptorMethodHandler	TemplatesImpl
Jdk7u21	LinkedHashSet	TemplatesImpl
Jython1	PriorityQueue	PyFunction
MozillaRhino1	BadAttributeValueExpException	TemplatesImpl
MozillaRhino2	NativeJavaObject	TemplatesImpl
Rome	HashMap	TemplatesImpl
Spring1	SerializableTypeWrapper	Runtime.exec
Spring2	SerializableTypeWrapper	Runtime.exec
Vaadin1	BadAttributeValueExpException	TemplatesImpl

This first analysis, illustrate well that in the subsets of known gadget chains analysed, only 4 out of 23 used a sink gadget other than *Runtime.exec* or *GetterMethodImpl*. Furthermore, *GetterMethodImpl* calls behind the scene *Runtime.exec* meaning that we only need one sanitizer to catch the exploit. This analysis is however biased in the way that it only analyses gadgets allowing remote code execution. There exist other gadgets in the Ysoserial collection[2] allowing for file upload or *JRMP* (Java Remote Method Protocol).

In the second phase of the analysis, I conducted a deeper analysis by identifying the different intermediate gadgets for 8 chains. For every gadget, I tried to categorise every gadget regarding its number of parameters. Here is the classification :

- Source Gadget: This starts off the chain with a custom readObject implementation
- Sink Gadget: This gadget triggers the dangerous functionality
- Identity Gadget: This gadget on being invoked returns the instantiated class member itself
- UnaryOp Gadget: This gadget takes a class member and calls a method from it

- BinaryOp Gadget: This gadget takes two class members and calls them with one as an argument for another
- NaryOp Gadget: This gadget takes n class members and calls them all

The very interesting point identified in this phase is that to trigger a *Runtime.exec* or *templateImpl.getOutputProperties*, the last gadget before the sink is a BinrayOp gadget that has the particularity of invoking a provided method name *invoke* on a provided object. Furthermore, the intermediate gadgets have the role to connect the source and the sink but do not have any particular pattern.

The main conclusion to this analysis is that there exist a few main sink gadgets (2/23) that are used in 90% of the cases, therefore external libraries are almost always used as a way to connect a source to a well-known sink. Out of the 23 chains, 13 starts with some basic collection that does not need any external library. If the serialization and deserialization process of those types were different and did not trigger any method on the deserialized object, then at least 50% of the gadgets chain analysed would be ineffective. Those first results are interesting but have been conducted on few data. It would be good to scale it up and see if the conclusion remains valid.

Chapter 4

Dynamic Analysis Module

This chapter will present the 2 main implementations done during this project, mainly the sanitizer using custom *hooks* and the generation of custom fake objects using *jazzer*.

4.1 Sanitizer

In the previous chapter, we presented the main gadgets identified, including some sink gadgets. Those specific gadgets, on which the exploit is based are very specific and in our analysis, were often identical (*Runtime.exec* and *TemplatesImpl.getOutputProperties*).

To detect the execution of the sink gadgets, we identified one main technical solution, the hooks offer by *jazzer*. It is possible to create some custom hooks that will get triggered during the execution. The creation of such a hook is as follows :

```java
package com.example;

import java.lang.Runtime;
import java.lang.Process;

import com.code_intelligence.jazzer.api.HookType;
import com.code_intelligence.jazzer.api.MethodHook;
import java.lang.invoke.MethodHandle;
import com.code_intelligence.jazzer.api.Jazzer;

public class RuntimeHook {
  @MethodHook(type = HookType.REPLACE, targetClassName = "java.lang.Runtime",
  targetMethod = "exec", targetMethodDescriptor = "")
  public static Process hookRuntime(MethodHandle handle, Object thisObject, Object[] args, int hookId) {
    Jazzer.reportFindingFromHook(new FuzzerSecurityIssueHigh("Runtime.exec triggered !"));
  }
}
```

Figure 4.1 – Runtime hook

However, during testing with exploits, it happen that some of our custom hooks were not called. It took some time to identify why but after some search it appears that the default behaviour of jazzer hook is to instrument callsites only. Therefore if you set up a hook on a methods like *Runtime.exec*, then the hook will be triggered only if the code that invokes this method is instrumented.

For example, *TemplatesImpl.getOutputProperties*) triggers *Runtime.exec* but not in an instrumented callsite therefore it was not possible to detect the exploit.

There exist, however, a solution to this problem, it is possible during instrumentation with *jazzer* to specify some additional classes to be instrumented using *-instrumentation_includes* or *-custom_hook_includes*. Once instrumented, hooks will be triggered every time the method is called even when the callsite is not instrumented. This allowed catching every exploit on the existing gadgets chain that served as a test bench.

4.2 Fake objects generation

The fuzzer we want to build is a bit different from the more traditional fuzzer. In traditional ones, we provide some input that needs to be handled correctly. In our case, gadgets correspond to method invocation that requires objects of the declaring class to be instantiated accordingly. If we detect potential gadget chains with specific objects then we can guide the generation of the chain by specifying the class of each object in the chain.

Jazzer, comes with a functionality named *autofuzz* that enables fuzzing arbitrary methods without having to manually create fuzz targets. Jazzer will attempt to generate suitable and varied inputs to a specified method using only public API functions available on the classpath. [1]

In their implementation, they have created a class named *Meta.java* that allow generating some custom objects using public methods. The way this class work is by first trying to generate some objects using constructors and setters and if it does not succeed, by searching for potential existing nested builders.

This class is a good basis for our fuzzer as it already provides the generation of some custom arbitrary classes, however, it has two main limitations :

- Might return null objects
- Do not allow to provide custom made sub-objects

The first limitation is the fact that it could return a null object. This is problematic because

[1] `https://github.com/CodeIntelligenceTesting/jazzer`

when we provide some objects to deserialize we do not want to provide a null value as it will not be able to trigger any deserialization exploit. The base object to deserialize should not be *null*.

The second issue is that the current implementation does not allow to provide existing objects as intermediary steps. For example, in our fuzzer, we would like to provide well-known sink gadgets that we are sure to be able to detect and that the fuzzer might not be able to generate easily. The idea is that the fuzzer will generate a path from the source to the sink.

To fix those limitations, I implemented a patch for this class. I added a default parameter *nullable = false* to specify if the object to generate could be null. Then added a new property *prebuiltObjects* of type *Map<class, List<Object»* that allows to provide some specific objects for a given class. When generating custom objects, there is a 50% chance of taking one of the provided prebuilt objects.

The main goal of the presented implementation was to be compatible with the existing code base and avoid any breaking change so that errors identified with another version would still trigger the issue.

Chapter 5

Related Work

During this project, we mainly identified and based our work on 2 existing projects.

The first one is Gadgets inspector, which is a tool allowing to generate a list of possible gadgets chains from a given list of the classpath. The main issue of this tool is the high rate of false-positive results in addition to the need to tweak the parameters to make the exploit work. [5]

The second one is the project *YsoSerial* which is a collection of identified deserialization gadgets chains. This dataset is useful to start with the understanding of the problem but is limited and do not allow any automatic discovery of gadget chains. [2]

There is, however, a few other projects that could be interesting to investigate :

- Joogle, Programatically query about types/methods on the classpath[9]
- Marshalsec, Deserialization payload generator for numerous libraries and gadget chains[7]

Chapter 6

Conclusion

Deserialization vulnerabilities are part of the top 10 OWASP[11] with potential impactful consequences like remote code execution. Currently, there exists only one way of identifying deserialization vulnerabilities (gadget inspector) but this tool produces high false positives that need to be checked manually.

The idea of this project is to set up the basic building block to the implementation of a fuzzer allowing to detect such types of vulnerabilities. The first building block is the analysis of known gadget chains that lead to the observation that external libraries are mostly used for linking existing source and sink gadgets (50% of the cases). And that 2 sink gadgets were used in 90% of gadget chains analysed.

The second block was the creation of a sanitizer to identify exploits and catch those. This step has been implemented using *Jazzer* and its hooks. I have been able to implement a proof of concept allowing me to catch every exploit of the existing gadget chains used.

The last building block implemented is the generation of fake objects, or more specifically, the extension of the functionality already implemented for the autofuzz tool of *Jazzer*. These improvements will make the generation of fake objects easier and will ease future development.

To conclude this project has allowed the development of some new building blocks that will be used for the implementation of the fuzzer.

Bibliography

[1] CVE. *CVE-2015-8103*. 2015. URL: https://www.cve.org/CVERecord?id=CVE-2015-8103 (visited on 01/01/2022).

[2] Frohoff. *YsoSerial*. 2015. URL: https://github.com/frohoff/ysoserial (visited on 01/01/2022).

[3] Frohoff and Lawrence. *AppSecCali 2015: Marshalling Pickles*. 2015. URL: https://frohoff.github.io/appseccali-marshalling-pickles/ (visited on 01/01/2022).

[4] Ian Haken. *Automated Discovery of Deserialization Gadget Chains*. 2018. URL: https://i.blackhat.com/us-18/Thu-August-9/us-18-Haken-Automated-Discovery-of-Deserialization-Gadget-Chains-wp.pdf (visited on 01/01/2022).

[5] Ian Haken. *Gadget Inspector*. 2018. URL: https://github.com/JackOfMostTrades/gadgetinspector (visited on 01/01/2022).

[6] Alexeu Kojenov. *Deserialization: what, how and why [not]*. 2018. URL: https://appsecus2018.sched.com/event/F04J (visited on 01/01/2022).

[7] mbechler. *Marshalsec*. 2017. URL: https://github.com/mbechler/marshalsec (visited on 01/01/2022).

[8] NIST. *CVE-2015-8103*. 2015. URL: https://nvd.nist.gov/vuln/detail/CVE-2015-8103 (visited on 01/01/2022).

[9] Contrast Security OSS. *Joogle*. 2016. URL: https://github.com/Contrast-Security-OSS/joogle (visited on 01/01/2022).

[10] OWASP. *A08 2021 - Software and Data Integrity Failures*. 2021. URL: https://owasp.org/Top10/A08_2021-Software_and_Data_Integrity_Failures/ (visited on 01/01/2022).

[11] OWASP. *OWASP Top 10*. 2021. URL: https://owasp.org/www-project-top-ten/ (visited on 01/01/2022).

[12] Marc Schönefeld. *Pentesting J2EE*. 2006. URL: https://www.blackhat.com/presentations/bh-federal-06/BH-Fed-06-Schoenefeld-up.pdf (visited on 01/01/2022).