# DÉSÉRIALISATION JAVA : UNE BRÈVE INTRODUCTION

Alexandre BARTEL – Jacques KLEIN – Yves LE TRAON
*Uni.lu / SnT*

**mots-clés :** ***EXPLOIT / JAVA / DÉSÉRIALISATION / EXÉCUTION DE CODE ARBITRAIRE / RCE***

*Certaines librairies Java permettent de transformer un objet en un flux d'octets et vice-versa. Ces processus sont appelés sérialisation et déserialisation, respectivement. Ces processus ne manipulent qu'un flux d'octets qui représente des données et non du code. Nous présentons dans cet article les bases de la sérialisation en Java et nous analysons une méthode présente dans les libraires standards de la machine virtuelle Java qui ouvre la porte à une vulnérabilité de désérialisation.*

## 1 Introduction

Dans de précédents articles de *MISC* **[1,2,3]**, nous avions détaillé plusieurs vulnérabilités Java permettant de contourner la sandbox Java en désactivant le manager de sécurité (`SecurityManager`). Pour exécuter du code arbitraire avec ces vulnérabilités, l'attaquant doit déjà pouvoir exécuter du code sur une machine virtuelle de sa cible. Cela n'est pas facile lorsque la cible est un « client », car les navigateurs web ne permettent plus par défaut d'exécuter des applets Java **[4,5]**. Cela devient difficile lorsque la cible est un « serveur », car il est rare qu'il exécute directement du code Java d'un client (il y a au moins une exception : Apache Spark **[6]**). Dans la plupart des cas, le code Java d'un serveur n'exécutera pas de code autre que celui des classes présentes sur le serveur et, à cause d'une fausse sensation de sécurité, les machines virtuelles auront une fâcheuse tendance à ne pas activer le manager de sécurité pour restreindre les permissions du code Java. Cependant, le serveur peut manipuler des données provenant d'un utilisateur et reconstruire des objets Java à partir de ces données. Dans cet article, nous analysons une méthode qui est le cœur d'une vulnérabilité de désérialisation présente dans une classe des librairies de la machine virtuelle Java.

Pour commencer, nous allons décrire le mécanisme de sérialisation de Java dans la section 2. Ensuite, dans la section 3, nous analyserons la méthode vulnérable.

## 2 Sérialisation et désérialisation en Java

### 2.1 La sérialisation en Java

Sérialiser un objet c'est le transformer en un tableau d'octets. Tous les langages orientés objet disposent de libraires pour sérialiser les objets : pickle pour Python **[7]**, boost pour C++ **[8]**, java.io pour Java **[9]**, etc. La sérialisation est utilisée principalement pour sauvegarder l'état d'objets localement ou pour transférer des objets entre programmes. Sauvegarder un objet peut être intéressant si la création de l'objet est gourmande en temps et mémoire (désérialiser l'objet est alors beaucoup plus rapide que de le recréer à chaque fois). Transférer un objet peut être nécessaire pour les appels de méthodes à distance par exemple (SOAP **[10]**, CORBA **[11]**, etc.).

En Java, pour être sérialisable, un objet doit implémenter l'interface `Serializable`. Ci-dessous se trouve un bout de code illustrant le processus de sérialisation. Ce programme va écrire un objet sérialisé de type `MaClasse` dans le fichier `/tmp/MaClasse.ser` :

```
public class MaClasse implements Serializable {

  int i;

  public MaClasse(int i) {
```

```
    this.i = i;
  }

  public static void main(String [] args) throws Throwable {
    MaClasse mc1 = new MaClasse(0xdeadbeef);
    FileOutputStream fos = new FileOutputStream(
      new File("/tmp/MaClasse.ser"));
    ObjectOutputStream oos = new ObjectOutputStream(fos);
    oos.writeObject(mc1);
    oos.close();
    fos.close();
  }
}
```

Le contenu du fichier **MaClasse.ser** est le suivant :

```
$ hexdump -C /tmp/MaClasse.ser
00000000  ac ed 00 05 73 72 00 08  4d 61 43 6c 61 73 73 65
|....sr..MaClasse|
00000010  d4 00 a2 90 63 60 89 b5  02 00 01 49 00 01 69 78
|....c`.....I..ix|
00000020  70 de ad be ef                                    |p....|
```

La structure d'un flux d'objets sérialisés est définie par une grammaire **[12]**. Le flux commence toujours par le nombre magique **0xaced**. Ensuite, il y a le numéro de version, **0x005**, puis un octet représentant le type de contenu (ici **0x73** pour un objet « normal ») puis un octet, **0x72**, indiquant que ce qui va suivre est une description de classe. La description de classes commence par le nom de la classe, précédée du nombre d'octets du nom. Soit **0x0008** pour le nombre d'octets, puis la chaîne de caractères « MaClasse ». Ensuite se trouve l'identifiant de sérialisation de la classe **0xd400a290636089b5** généré par défaut à partir des caractéristiques de la classe qui peuvent dépendre du compilateur **[9]**. Puis, il y a un octet avec la valeur **0x02** qui indique que la classe implémente **Serializable** (d'autres classes implémentant **Enum** ou **Externalizable** peuvent aussi être sérialisées). Ensuite, **0x0001** indique le nombre de champs de la classe. Il y a donc un champ. L'octet suivant, **0x49**, indique qu'il s'agit d'un entier. Puis il y a le nom de l'entier en UTF-8 **[13]** : **0x000169** (« **i** ») suivit de l'octet **0x78** qui indique la fin du bloc de données, puis de l'octet **0x70** qui indique que la superclasse est **java.lang.Object**. Finalement, **0xdeadbeef** indique la valeur du champ **i**.

À travers cet exemple, nous remarquons que seules la description des classes et les données des champs sont sérialisées. En d'autres termes, le code des méthodes n'est pas sérialisé et il faut donc que la machine virtuelle Java (JVM) qui désérialise ait exactement les mêmes classes (la même description et le même identifiant de sérialisation) dans son **classpath** que la JVM qui sérialise. De la même manière qu'un attaquant chaîne des gadgets (bouts de code présents dans le processus cible) pour effectuer une attaque ROP, un attaquant exploitant une vulnérabilité de désérialisation Java va chaîner des méthodes Java de classes présentes dans le **classpath** de la JVM qui fait tourner le processus qui désérialise le flux d'octets contrôlé par l'attaquant.

## 2.2 La désérialisation en Java

La désérialisation est l'étape opposée à la sérialisation c'est-à-dire que des objets sont créés à partir d'un tableau d'octets. Comme illustré ci-dessous, un objet est reconstruit via l'appel à la méthode **readObject()** sur une instance d'**ObjectInputStream** :

```
public class Lecteur {

  public static void main(String[] args) throws Throwable {
    FileInputStream fis = new FileInputStream(new File("/tmp/
MaClasse.ser"));
    ObjectInputStream ois = new ObjectInputStream(fis);
    MaClasse mc = (MaClasse) ois.readObject();
    System.out.println("mc.i = 0x" + Integer.toHexString(mc.i));
  }

}
```

En exécutant ce code, l'instance de **MaClasse** est bien reconstruite à partir de la version sérialisée présente dans le fichier **/tmp/MyClasse.ser** et la valeur du champ **i** est bien la valeur de l'instance lors de la sérialisation :

```
$ java Lecteur
mc.i = 0xdeadbeef
```

Pour reconstruire l'objet, la méthode **readObject()** n'utilise pas le constructeur de **MaClasse**, mais initialise les champs de la classe à reconstruire (ici uniquement **i**) à partir des valeurs présentes dans le flux d'octets de l'objet sérialisé. Jusqu'à présent tout va bien, l'attaquant ne peut pas exécuter de code. Cependant, ce processus de désérialisation n'est pas suffisant pour désérialiser des objets plus complexes comme **java.util.HashMap**. En effet, pour équilibrer une table de hachage qui est représentée par une séquence d'alvéoles qui contiennent des paires clé-valeur, l'alvéole dans laquelle va résider la paire est déterminée en fonction de la valeur de hachage de la clé. Or, pour une clé donnée, il n'est pas garanti que sa valeur de hachage soit la même entre deux implémentations de la JVM ou entre deux exécutions du programme **[14]**. La solution est de personnaliser le processus de sérialisation et de désérialisation pour ce type d'objets.

## 2.3 Personnalisation

Reprenons l'exemple de la table de hachage qui a les champs non-statiques suivants :

```
public class HashMap<K,V>
    extends AbstractMap<K,V>
    implements Map<K,V>, Cloneable, Serializable
{

    transient Entry<K,V>[] table;
    transient int size;
    int threshold;
    final float loadFactor;
    transient int modCount;
```

```
    transient boolean useAltHashing;
    transient final int hashSeed = sun.misc.Hashing.
randomHashSeed(this);
[...]
}
```

Les champs marqués **transient** ne seront pas sérialisés par défaut. Pour personnaliser la désérialisation, la classe **HashMap** implémente la méthode **writeObject()** suivante (appelée par **ObjectOutputStream.writeObject(Object o)**) :

```
    private void writeObject(java.io.ObjectOutputStream s)
        throws IOException
    {
        Iterator<Map.Entry<K,V>> i =
            (size > 0) ? entrySet0().iterator() : null;

        s.defaultWriteObject();

        s.writeInt(table.length);

        s.writeInt(size);

        if (size > 0) {
            for(Map.Entry<K,V> e : entrySet0()) {
                s.writeObject(e.getKey());
                s.writeObject(e.getValue());
            }
        }
    }
```

Cette méthode va tout d'abord appeler **defaultWriteObject()** pour sérialiser les champs non **transient threshold** et **loadFactor**. Ensuite, elle va forcer la sérialisation des champs **table.length** et **size**. Finalement, si la table de hachage n'est pas vide, elle va écrire les paires clé-valeur les unes à la suite des autres. Les champs **modCount**, **useAltHashing** et **hashSeed** n'ont pas été sérialisés et devraient être initialisés lors de la désérialisation. La table de hachage va être reconstruite via l'implémentation de la méthode **readObject()** suivante (appelée par **ObjectInputStream.readObject()**) :

```
    private void readObject(java.io.ObjectInputStream s)
        throws IOException, ClassNotFoundException
    {
       // Read in the threshold (ignored), loadfactor, and any hidden
stuff
       s.defaultReadObject();
       if (loadFactor <= 0 || Float.isNaN(loadFactor))
           throw new InvalidObjectException("Illegal load factor: " +
                                            loadFactor);

       // set hashSeed (can only happen after VM boot)
       Holder.UNSAFE.putIntVolatile(this, Holder.HASHSEED_OFFSET,
               sun.misc.Hashing.randomHashSeed(this));

       // Read in number of buckets and allocate the bucket array;
       s.readInt(); // ignored

       // Read number of mappings
       int mappings = s.readInt();
       if (mappings < 0)
           throw new InvalidObjectException("Illegal mappings count: " +
                                            mappings);
       int initialCapacity = (int) Math.min(
               // capacity chosen by number of mappings
               // and desired load (if >= 0.25)
               mappings * Math.min(1 / loadFactor, 4.0f),
               // we have limits...
               HashMap.MAXIMUM_CAPACITY);
       int capacity = 1;
       // find smallest power of two which holds all mappings
       while (capacity < initialCapacity) {
           capacity <<= 1;
       }

       table = new Entry[capacity];
       threshold = (int) Math.min(capacity * loadFactor, MAXIMUM_
CAPACITY + 1);
       useAltHashing = sun.misc.VM.isBooted() &&
               (capacity >= Holder.ALTERNATIVE_HASHING_THRESHOLD);

       init();  // Give subclass a chance to do its thing.

       // Read the keys and values, and put the mappings in the
HashMap
       for (int i=0; i<mappings; i++) {
           K key = (K) s.readObject();
           V value = (V) s.readObject();
           putForCreate(key, value);
       }
   }
```

L'objet représentant la table de hachage est initialisé avec l'appel vers **readDefaultObject()**, puis les champs **transient** sont initialisés. Enfin, les paires clé-valeur sont insérées dans la table via la boucle **for** qui lit chaque paire puis l'insère dans la table de hachage.

Il y a maintenant exécution de code. En effet, la méthode **putForCreate()**, va potentiellement manipuler des objets créés à partir de données contrôlées par l'attaquant. Manipuler un objet signifie lire/écrire des champs de l'objet ou exécuter des méthodes sur l'objet. Un attaquant va donc examiner les classes sérialisables qui personnalisent la méthode **readObject()** pour trouver des points d'entrée pour son attaque.

## 3 Vulnérabilité dans les classes de la JVM

La vulnérabilité présentée ici est le fruit du travail de Chris Frohoff **[15]**. La vulnérabilité – présente depuis les versions 1.6 jusqu'à la version 1.7 update 21 de la JVM – permet à un attaquant d'exécuter du code arbitraire si le programme cible désérialise un flux d'octets contrôlé par l'attaquant. Pour que l'attaque fonctionne, il faut : (1) trouver une méthode « intéressante », **CL.m()**, qui va permettre d'exécuter du code contrôlé par l'attaquant (voir ci-dessous) et (2) trouver s'il existe un chemin d'exécution depuis **readObject()** jusqu'à **CL.m()**. Par manque de place, nous aborderons le point (2) dans un prochain article.

Dans la librairie Java (Java Class Library ou JCL) – les classes présentes par défaut dans le **classpath** de la JVM – se trouve la classe **com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl**. Cette classe est sérialisable et définit la méthode **getTransletInstance()** dont le code est le suivant :

```
   private Translet getTransletInstance()
       throws TransformerConfigurationException {
       try {
           if (_name == null) return null;

           if (_class == null) defineTransletClasses();

           // The translet needs to keep a reference to all its auxiliary
           // class to prevent the GC from collecting them
           AbstractTranslet translet = (AbstractTranslet) _class[_
transletIndex].newInstance();
           translet.postInitialization();
           translet.setTemplates(this);
           translet.setServicesMechnism(_useServicesMechanism);
           if (_auxClasses != null) {
               translet.setAuxiliaryClasses(_auxClasses);
           }

           return translet;
       }
       catch (InstantiationException e) {
           ErrorMsg err = new ErrorMsg(ErrorMsg.TRANSLET_OBJECT_ERR, _name);
           throw new TransformerConfigurationException(err.toString());
       }
       catch (IllegalAccessException e) {
           ErrorMsg err = new ErrorMsg(ErrorMsg.TRANSLET_OBJECT_ERR, _name);
           throw new TransformerConfigurationException(err.toString());
       }
   }
```

Cette méthode va appeler **`newInstance()`** sur un élément du tableau **`_class`**. Comme **`_class`** est un champ de la classe, il est sous le contrôle de l'attaquant – même s'il est privé – car l'attaquant peut le modifier via le moteur de réflexion Java. De la même manière, **`_transletIndex`** est aussi sous le contrôle de l'attaquant. Ça fait une belle jambe à l'attaquant : il peut maintenant potentiellement instancier n'importe quelle classe présente dans le **`classpath`**. Pour ce faire, il faudrait que la méthode **`getTransletInstance()`** soit appelée depuis une méthode **`readObject()`** directement. Malheureusement, ce n'est pas le cas. De plus, en créant une nouvelle classe, l'attaquant ne pourrait qu'exécuter le code d'une méthode d'initialisation (**`<clinit>`** générée à la compilation) de n'importe quelle classe présente dans la JCL. Cela n'est pas forcément très utile pour exécuter du code arbitraire, car les méthodes d'initialisation de classe servent principalement à initialiser les champs statiques. Rien n'est perdu, car la méthode **`getTransletInstance()`** réserve une autre surprise… Il se trouve que cette méthode appelle **`defineTransletClasses()`** si **`_class`** est *null*. Cette méthode est représentée ci-dessous :

```
    private void defineTransletClasses()
        throws TransformerConfigurationException {

        if (_bytecodes == null) {
            ErrorMsg err = new ErrorMsg(ErrorMsg.NO_TRANSLET_CLASS_ERR);
            throw new TransformerConfigurationException(err.toString());
        }

        TransletClassLoader loader = (TransletClassLoader)
            AccessController.doPrivileged(new PrivilegedAction() {
                public Object run() {
                    return new TransletClassLoader(ObjectFactory.
findClassLoader());
                }
            });

        try {
            final int classCount = _bytecodes.length;
```

```
            _class = new Class[classCount];

            if (classCount > 1) {
                _auxClasses = new Hashtable();
            }

            for (int i = 0; i < classCount; i++) {
                _class[i] = loader.defineClass(_bytecodes[i]);
                final Class superClass = _class[i].getSuperclass();

                // Check if this is the main class
                if (superClass.getName().equals(ABSTRACT_TRANSLET)) {
                    _transletIndex = i;
                }
                else {
                    _auxClasses.put(_class[i].getName(), _class[i]);
                }
            }

            if (_transletIndex < 0) {
                ErrorMsg err= new ErrorMsg(ErrorMsg.NO_MAIN_TRANSLET_ERR,
_name);
                throw new TransformerConfigurationException(err.toString());
            }
        }
        catch (ClassFormatError e) {
            ErrorMsg err = new ErrorMsg(ErrorMsg.TRANSLET_CLASS_ERR, _name);
            throw new TransformerConfigurationException(err.toString());
        }
        catch (LinkageError e) {
            ErrorMsg err = new ErrorMsg(ErrorMsg.TRANSLET_OBJECT_ERR, _name);
            throw new TransformerConfigurationException(err.toString());
        }
    }
```

Cette méthode va définir une nouvelle classe à partir du tableau d'octets `_bytecode` qui est aussi sous le contrôle de l'attaquant ! La classe ainsi définie sera insérée dans le tableau `_class` à l'indice 0. Le champ `_transletIndex` sera lui aussi initialisé à 0. Maintenant, c'est game over : l'attaquant peut (1) initialiser `_bytecode` avec un tableau d'octets représentant une classe contrôlée par l'attaquant (2) charger cette classe dans la JVM (via la méthode `defineClass()`) et (3) créer une instance de cette classe (via la méthode `newInstance()`) ! Il peut maintenant exécuter du code arbitraire, car lors de la création de l'instance, la méthode `<clinit>()` – de la classe que l'attaquant contrôle – va être exécutée. Il ne lui reste plus qu'à trouver une combinaison de méthodes pour atteindre la méthode `getTransletInstance()` depuis `readObject()`.

## Conclusion

Si la sérialisation Java part d'une bonne intention, i.e., faciliter la vie du programmeur, en pratique elle permet à un attaquant de contrôler toutes les données d'une classe à désérialiser. Comme nous l'avons vu, certaines classes comportent des méthodes « dangereuses » qui utilisent ces données – potentiellement contrôlées par un attaquant – pour définir de nouvelles classes... Nous verrons dans un prochain article comment combiner des méthodes Java pour parvenir à atteindre le code de ces méthodes « dangereuses » et ainsi exécuter du code arbitraire sur la machine cible. Nous présenterons aussi les principales approches pour nous prémunir des attaques de désérialisation en Java. ■

## Références

[1] Alexandre Bartel. *Un cas d'école de contournement d'ASLR avec le CVE-2010-0840*, Multi-System & Internet Security Cookbook (MISC), 89:4–11, 2017

[2] Alexandre Bartel. *Exploitation du CVE-2015-4843*. Multi-System & Internet Security Cookbook (MISC), 95:4–9, 2018

[3] Alexandre Bartel, Jacques Klein et Yves Le Traon. *Exploitation du CVE-2017-3272*. Multi-System & Internet Security Cookbook (MISC), 97:14-17, 2018

[4] *The final countdown for npapi*, Google, https://blog.chromium.org/2014/11/the-final-countdown-for-npapi.html, 2014

[5] *Npapi plugins in Firefox*, Mozilla, https://blog.mozilla.org/futurereleases/2015/10/08/npapi-plugins-in-firefox/, 2015

[6] *Apache Spark, a unified analytics engine for large-scale data processing*, The Apache Software Foundation, http://spark.apache.org/

[7] *Python object serialization*, The Python Software Foundation, https://docs.python.org/2/library/pickle.html

[8] *Serialization*, Robert Ramey, https://www.boost.org/doc/libs/1_60_0/libs/serialization/doc/index.html

[9] *Serializable*, Oracle, https://docs.oracle.com/javase/10/docs/api/java/io/Serializable.html

[10] *Simple Object Access Protocol (SOAP) for Java*, Oracle, https://docs.oracle.com/cd/A97630_01/appdev.920/a96616/arxml11.htm

[11] *CORBA Technology and the Java™ Platform Standard Edition*, Oracle, https://docs.oracle.com/javase/7/docs/technotes/guides/idl/corba.html

[12] https://docs.oracle.com/javase/7/docs/platform/serialization/spec/protocol.html

[13] *What is the definition of UTF-8 ?*, Unicode Inc., http://unicode.org/faq/utf_bom.html#utf8-1

[14] Bloch, Joshua. *Effective java (the java series)*. Prentice Hall PTR, 2008.

[15] Chris Frohoff, *Unsafe Object Deserialization Security Advisory* – Java SE https://gist.github.com/frohoff/24af7913611f8406eaf3

[16] Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides, *Design Patterns : Elements of Reusable Object-Oriented Software*, Addison-Wesley, ISBN 0-201-63361-2, p233-245, http://www.uml.org.cn/c++/pdf/DesignPatterns.pdf

[17] Proxy, Oracle, https://docs.oracle.com/javase/7/docs/api/java/lang/reflect/Proxy.html

[18] Éric Bruneton, Lenglet, R., Coupaye, T. : *ASM : a code manipulation tool to implement adaptable systems* (2002). http://asm.ow2.org/current/asm-eng.pdf

[19] *Serial Killer : Silently Pwning Your Java Endpoints*, Alvaro Muñoz, Christian Schneider, RSA 2016, https://www.rsaconference.com/writable/presentations/file_upload/asd-f03-serial-killer-silently-pwning-your-java-endpoints.pdf