DE L'UTILISATION D'UNE BIBLIOTHÈQUE À L'EXÉCUTION D'UN CODE ARBITRAIRE

Imen SAYAR & **Alexandre BARTEL**

Dans cet article, nous présentons une vulnérabilité de la version 3.1 de Commons Collections. Cette vulnérabilité, nommée « CommonsCollections1 », permet à un attaquant l'exécution d'un code arbitraire ou Remote Code Execution (RCE). Ce travail reprend certains concepts des deux articles publiés dans les versions précédentes de MISC en 2018 et 2019 [1,2].

mots-clés : *VULNÉRABILITÉ / DÉSÉRIALISATION EN JAVA / APACHE COMMONS COLLECTIONS / EXÉCUTION D'UN CODE ARBITRAIRE (RCE)*

1. INTRODUCTION

La sérialisation en Java permet de transformer des instances d'objets en un flux d'octets. Les objets peuvent donc être transférés à travers un réseau pour, par exemple, invoquer des méthodes à distance. Les objets peuvent aussi être stockés dans un fichier. Cela est intéressant dans le cas où un objet prend du temps à être construit. Il sera plus rapide de le construire une fois puis de le stocker et de le lire à partir du fichier à chaque exécution que de le reconstruire dynamiquement à chaque exécution. La désérialisation, processus inverse de la sérialisation, présente un danger quand le fichier sérialisé peut être généré par un attaquant. Ce danger n'épargne pas les applications les plus « sécurisées » telles que celles de PayPal **[6]** et de l'Agence de transport métropolitain de San Francisco (SFMTA) **[7]**. La première application a présenté une faille de sécurité - découverte en interne par Michael Stepankin en décembre 2015 - qui permet à un attaquant d'exécuter une commande shell et d'accéder aux bases des données de PayPal à partir d'un de ses sites web, **manager.paypal.com**. La deuxième application a été exploitée en novembre 2016 et a permis à un attaquant d'avoir accès aux ordinateurs de la SFMTA, les infectant via l'exploitation d'une vulnérabilité de désérialisation d'un serveur Oracle WebLogic. Ces deux vulnérabilités ont été causées par l'utilisation d'une version vulnérable de la bibliothèque Java Commons Collections **[5]**.

2. VULNÉRABILITÉ COMMONSCOLLECTIONS1 (CVE 2015-7501)

2.1 Ysoserial

Nous utilisons cet outil, développé par Chris Frohoff **[4]** probablement lors d'un petit déjeuner, pour générer un fichier sérialisé contenant des données permettant d'exécuter une calculatrice « calc.exe ». Ce fichier est appelé « CommonsCollect1Vulnerable ». L'ensemble des classes du fichier sérialisé est représenté par la Figure 1. Dans cette figure, chaque champ intéressant d'une classe est précédé par le

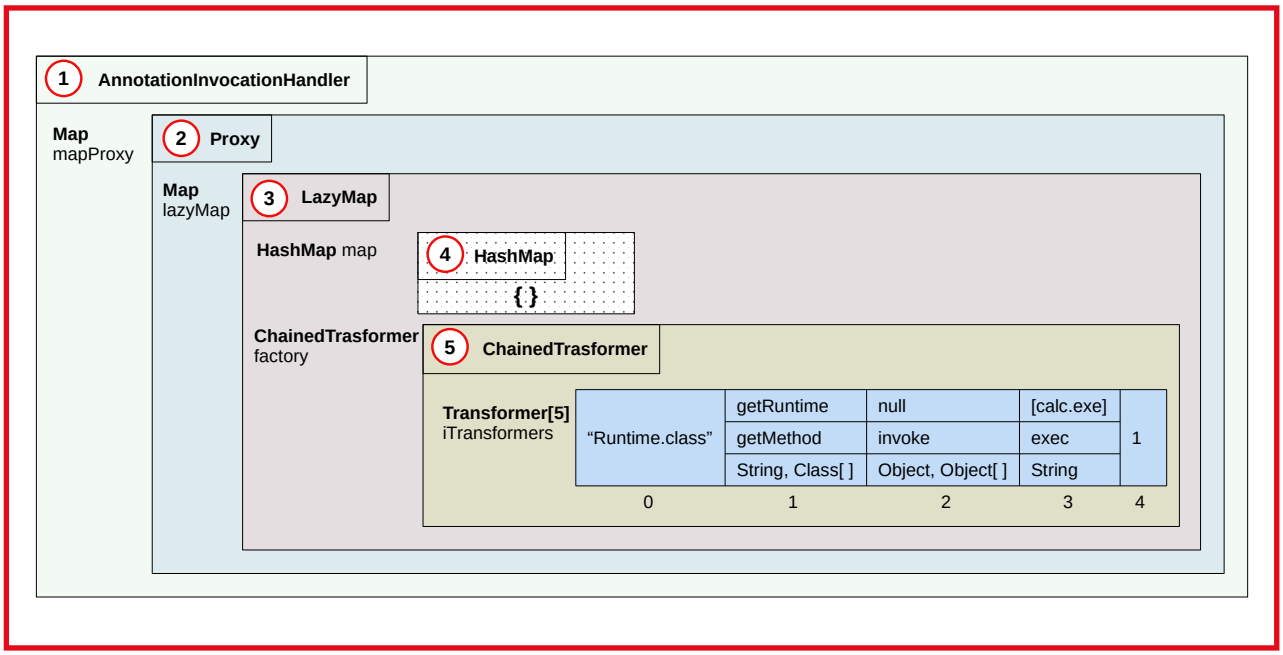

Fig. 1 : Vue d'ensemble sur la structure du fichier malveillant.

nom de son type déclaré dans le programme et suivi par un rectangle indiquant son type concret et son contenu. Par exemple, le champ **`mapProxy`** est précédé par son type **`Map`** tel qu'il est déclaré dans le programme malveillant de ysoserial. Le type concret de ce champ est **`Proxy`**.

2.2 Aperçu de l'attaque

Pour reproduire l'attaque, munissez-vous d'un bol, d'une petite cuillère, du lait, d'un paquet de céréales et suivez les instructions suivantes :

1. générer un fichier sérialisé malveillant en utilisant par exemple ysoserial en lui indiquant que la charge consiste en l'exécution de la calculatrice ;
2. créer une classe « victime » qui désérialise le fichier malveillant généré en 1. Cette désérialisation s'effectue en utilisant la méthode **`readObject()`** d'un objet **`ObjectInputStream`** permettant de lire le flux d'octets du fichier en entrée ;
3. rajouter la bibliothèque vulnérable Apache Commons Collections 3.1.jar dans le classpath de la classe victime ;
4. exécuter la classe victime.

Notons que même si le programme n'utilise pas directement les classes vulnérables, il est toujours vulnérable si ces classes sont dans son classpath. Dans **[1]**, les auteurs fournissent un programme qui désérialise des données de source inconnue en utilisant la méthode **`readObject()`**. Dans le cas du présent article, ces données sont contenues dans le fichier « CommonsCollect1Vulnerable » généré par ysoserial. Il est à noter que, parmi les données contenues dans ce fichier malveillant, il y a deux instances d'**`AnnotationInvocationHandler`** : la première concerne un **`Proxy`** (numéro 1 dans la figure 1) et la deuxième concerne une table de hachage, **`LazyMap`** (numéro 3 de la même figure). La manipulation de ces données est expliquée dans la suite de cet article.

La désérialisation redirige l'exécution vers des appels de quelques méthodes donnant la main à l'attaquant pour contrôler certains champs et exécuter son code arbitraire. La pile d'appels de l'attaque est la suivante :

```
(23) Runtime.exec(String[], String[], File) line: 615
(22) Runtime.exec(String, String[], File) line: 448
(21) Runtime.exec(String) line: 345
(20) NativeMethodAccessorImpl.invoke0(Method, Object,
Object[]) line: not available [native method]
(19) NativeMethodAccessorImpl.invoke(Object, Object[])
line: 57
```

```
(18) DelegatingMethodAccessorImpl.invoke(Object,
Object[]) line: 43
(17) Method.invoke(Object, Object...) line: 601
(16) InvokerTransformer.transform(Object) line: 125
(15) ChainedTransformer.transform(Object) line: 122
(14) LazyMap.get(Object) line: 151
(13) AnnotationInvocationHandler.
invoke(Object,Method,Object[]) line: 69
(12) $Proxy0.entrySet() line: not available
(11) AnnotationInvocationHandler.
readObject(ObjectInputStream) line: 346
(10) NativeMethodAccessorImpl.invoke0(Method, Object,
Object[]) line: not available [native method]
(9) NativeMethodAccessorImpl.invoke(Object, Object[])
line: 57
(8) DelegatingMethodAccessorImpl.invoke(Object,
Object[]) line: 43
(7) Method.invoke(Object, Object...) line: 601
(6) ObjectStreamClass.invokeReadObject(Object,
ObjectInputStream) line: 1004
(5) ObjectInputStream.readSerialData(Object,
ObjectStreamClass) line: 1891
(4) ObjectInputStream.readOrdinaryObject(boolean) line:
1796
(3) ObjectInputStream.readObject0(boolean) line: 1348
(2) ObjectInputStream.readObject() line: 370
(1) ApplicationCibleVulnerable.main(String[]) line: 19
```

2.3 Machine virtuelle de Java (JVM)

Une application est potentiellement vulnérable si elle désérialise depuis une source non digne de confiance. Dans notre cas, cette application est représentée par **ApplicationCibleVulnerable**, voir la pile d'appels présentée dans 2.2. Dans **main**, l'application va désérialiser le flux d'octets. Les appels de (2) à (11) représentent le fonctionnement interne de la JVM qui va chercher à appeler la méthode **readObject** sur le premier objet à désérialiser qui, dans notre cas, est de type **AnnotationInvocationHandler**. Le pseudo-code de cette méthode est le suivant :

```
private void readObject(final ObjectInputStream
objectInputStream) {
     objectInputStream.defaultReadObject();
     AnnotationType instance;
     ...
     final Map<String, Class<?>> memberTypes =
instance.memberTypes();
     for (final Map.Entry<String, Object> entry :
this.memberValues.entrySet()) {
       [...]
     }
   }
```

La valeur de **this.memberValues** dans notre cas est **$Proxy0**. Dans la suite, nous expliquerons ce qu'est un proxy ainsi que le reste de la pile d'appels de l'attaque.

3. CHEMIN ENTRE $PROXY0.ENTRYSET() ET RUNTIME.EXEC()

Nous avons vu qu'à partir d'un appel de la méthode **readObject()** sur un objet de source inconnue, nous arrivons à rediriger les appels vers **$Proxy0.entrySet()**.

3.1 Proxy

Un **Proxy** en Java est une classe générée lors de l'exécution pour implémenter des interfaces. Il est associé avec un gestionnaire d'invocation représenté par la classe **InvocationHandler**. La JVM utilise la réflexion pour rediriger tout appel de méthode sur un **Proxy** vers la méthode **invoke()** de l'interface implémentée par ce **Proxy**. Ceci explique le passage de l'appel **$Proxy0.entrySet()** (appel numéro 12 de la pile) vers l'appel d'**AnnotationInvocationHandler.invoke(Object,Method,Object[])** (appel numéro 13). Ici, l'appel de cette méthode s'effectue sur une première instance d'**AnnotationInvocationHandler** créée dans le fichier sérialisé de l'attaquant. Notons que le type déclaré du proxy **mapProxy** est **Map** (qui est une interface) et son type concret est **Proxy**. D'après la spécification en **[9]**, le type du proxy est le type de l'interface qu'il implémente (dans notre cas **Map**).

```
1 public Object invoke(final Object o, final Method
method, final Object[] array) {
2 final String name = method.getName();
3 final Class<?>[] parameterTypes = method.
getParameterTypes();
4 if (name.equals("equals") && parameterTypes.
length == 1 && parameterTypes[0] == Object.class) {
5 return this.equalsImpl(array[0]);
6 }
10 [...]
11 Object o2 = this.memberValues.get(name);
12 [...]
13 return o2;
14 }
```

Dans le pseudo-code ci-dessus, la méthode **invoke()** prend en entrée trois arguments :

- un objet **o** de valeur **$Proxy0** ;
- une méthode **method** de valeur **Map.entrySet()** ;
- un tableau d'objets **array** ne contenant aucun élément (valeur **null**).

La variable **name** (ligne 2) prend le nom de la méthode **Map.entrySet()** passée en argument. Elle a comme valeur **entrySet**. La méthode **invoke()** effectue différentes comparaisons entre **name** et des chaînes de caractères afin d'identifier quelle méthode il faut appeler. Lors de la création d'un objet **AnnotationInvocationHandler**, il y a un champ appelé **memberValues** initialisé par la valeur **LazyMap**. C'est à ce moment de désérialisation que la **LazyMap**, une classe de Commons Collections, est créée. Ceci est dû à la récupération de la deuxième instance d'**AnnotationInvocationHandler** du fichier malveillant (la partie numéro 3 de la figure 1).

Par la suite, un objet **o2** est initialisé à **this.memberValues.get(name)** (ligne 11) qui est égal à **LazyMap.get(entrySet)**, car **this.memberValues** est une **LazyMap**. Cet objet **o2** est retourné (ligne 13) produisant l'appel de la méthode **get(Object)** de **LazyMap** (appel 14 de la pile).

3.2 LazyMap

Une **LazyMap** est une table de hachage contenant des clés et leurs valeurs associées. Elle permet de décorer une autre carte pour créer des objets sur la carte à la demande. Lorsque sa méthode **get(Object)** est appelée avec une clé qui n'existe pas dans la carte, la fabrique **factory** est utilisée pour créer l'objet. L'objet créé sera ajouté à la carte à l'aide de la clé demandée **[3]**.

Le code Java correspondant à la méthode **get(Object)** est :

```
public Object get(Object key) {
     if (map.containsKey(key) == false) {
       Object value = factory.transform(key);
       map.put(key, value);
       return value;
     }
     return map.get(key);
   }
```

L'objectif de cette méthode est de retourner la valeur associée à l'objet **key** si cette valeur existe. À défaut, la méthode crée une valeur **value** à associer à **key**, intègre le couple (**key**, **value**) dans la carte **map** et retourne **value**.

Revenons à l'appel de **LazyMap.get(Object)** (appel 14 de la pile). La carte **map** - qui est de type **HashMap<K,V>** - est vide. Elle ne contient aucun élément, donc la clé **key**, ayant comme valeur **entrySet**, n'est attribuée à aucune valeur. Par conséquent, la condition **if (map.containsKey(key) == false)** est évaluée à **true** et le bloc d'instructions qui suit cette condition est exécuté. Sa première instruction **Object value = factory.transform(key)** fait appel à **ChainedTransformer.transform(key)** vu que **factory** est un **ChainedTransformer**, ou suite de **Transformer**, extrait du fichier malveillant. Le code ci-dessous présente un exemple de plusieurs transformateurs introduits dans un tableau nommé *transformers*. Mais, qu'est-ce qu'un **Transformer** ?

```
final Transformer[] transformers = new Transformer[] {
                new ConstantTransformer(field1),
                new InvokerTransformer("field2", new
Class[] {... }, new Object[] {
                    "field2_value", new Class[0] })
};
```

3.3 Transformers

Un **Transformer** est une interface de la bibliothèque Commons Collections implémentée par une ou plusieurs autres classes. Un transformateur est initialisé via son constructeur. Il est utilisé en appelant sa méthode **Object transform(Object o)** qui transforme l'objet **o** en entrée en un objet à la sortie. Parmi ces classes, il y a **ConstantTransformer** et **InvokerTransformer**.

3.3.1 ConstantTransformer

Cette implémentation du transformateur permet de retourner toujours la même constante peu importe l'objet en entrée.

3.3.2 InvokerTransformer

Ce transformateur est initialisé avec le nom d'une méthode **M** et ses paramètres **P**. Cette méthode **M** sera appelée via le moteur de réflexion sur l'objet **o** passé en paramètre de **transform** : **R = o.M(P)**. La méthode **transform** retourne ensuite l'objet **R**. Afin de mieux comprendre le fonctionnement de cette structure, prenons un exemple. Soit **A** une classe et **m(int a, int b)** une de ses méthodes :

```
public class A {
     public A () { } // constructeur
     public int m (int a, int b) {
         return a + b;
     }
}
```

L'utilisation du **InvokerTransformer** se réalise comme suit :

```
A instA = new A(); // nouvel objet A
Object[] obj = new Object[] {3,9}; // arguments de m
Class[] cl = new Class[] {int.class, int.class}; // types
des arguments de m
InvokerTransformer invTraM = new InvokerTransformer("m",
cl, obj);
Object o = invTraM.transform(instA); // appel de
transform
```

Tout d'abord, il faut créer une instance **instA** de **A**. Ensuite, il faut récupérer - dans le tableau **obj[]** - les valeurs des arguments passés dans la méthode **m** (ici 3 et 9) ainsi que leurs types dans un tableau **cl[]** (tous les deux arguments sont de type **int**). Puis, il faut créer une instance de la classe **InvokerTransformer** en donnant comme paramètres du constructeur le nom de la méthode à traiter (**m**), les types des paramètres de cette méthode (**cl**) et les arguments effectifs de la méthode (**obj**). Enfin, il faut appeler la méthode **transform** sur l'instance de la classe A. Cet appel invoque par réflexion la méthode **m()** sur l'instance de **A**.

3.3.3 ChainedTransformer

C'est une classe qui implémente l'interface **Transformer** et qui permet d'enchaîner des transformateurs. Elle est caractérisée par une méthode **transform(Object object)** dont le code est le suivant :

```
public Object transform(Object object) {
     for (int i = 0; i < iTransformers.length; i++) {
       object = iTransformers[i].transform(object);
     }
     return object;
   }
```

L'objet en entrée de cette méthode est passé au premier transformateur **iTransformer[0]**. Le résultat de sa transformation est passé comme entrée au deuxième transformateur et ainsi de suite.

3.4 Retour à la pile d'appels

Revenons sur le contenu du fichier malveillant généré par ysoserial. Il contient un tableau de cinq **Transformers** nommé **iTransformers** (voir la partie numéro 5 de la figure 1). Ces cinq éléments sont tous contrôlés par l'attaquant vu qu'il peut introduire les noms des champs dans les constructeurs de chacun des transformateurs du tableau. Avec ysoserial, l'attaquant introduit :

1. une **ConstantTransformer** initialisée avec la constante **Runtime.class** ;
2. un **InvokerTransformer** initialisé avec **"getMethod("getRuntime")"** comme nom de la méthode à appeler via réflexion ;
3. un **InvokerTransformer** initialisé avec **"invoke"** comme nom de la méthode à appeler via réflexion ;
4. un **InvokerTransformer** initialisé avec **"exec"** pour nom de méthode ;
5. une **ConstantTransformer** initialisée avec la constante **1**.

Dans la pile d'appels de l'attaque, l'appel de la méthode **LazyMap.get(key)** entraîne celui de **ChainedTransformer.transform(key)** avec **"entrySet"** comme valeur de **key**. Cette valeur peut être remplacée par n'importe quelle chaîne de caractères, **"pain au chocolat"** par exemple. Cinq itérations auront lieu lors de l'appel de la méthode **transform(..)** sur les transformateurs de ce **ChainedTransformer**. Le premier transformateur **ConstantTransformer** retourne une constante égale à **java.lang.Runtime.class**. Le deuxième transformateur prend cette constante en entrée et retourne une référence (de type **Method**) vers la méthode **java.lang.Runtime.getRuntime()**. En effet, ce transformateur appelle, via réflexion, l'instruction suivante : **Method m = java.lang.Runtime.class.getMethod("getRuntime")**. Le troisième transformateur transforme cette référence en **Runtime** qui est une instance de la classe **Runtime**. En effet, la méthode **static getRuntime()** permet de récupérer une instance de **Runtime** afin d'utiliser cette instance pour appeler la méthode **exec** (qui n'est pas statique). L'appel de **InvokerTransformer.transform(..)** sur l'instance **java.lang.Runtime** provoque l'invocation de la méthode **exec** par réflexion. Dans le corps de la

méthode `InvokerTransformer.transform(..)` de ce quatrième transformateur, l'instruction `method.invoke(input, iArgs)` redirige vers l'appel à `Method.invoke(Object, Object...)` vu que :

- `method` est de type et de valeur `exec` de la classe `Runtime` ;
- `input` a comme valeur l'instance de la classe `java.lang.Runtime` ;
- `iArgs` est la chaîne de caractères `calc.exe`.

Comme expliqué précédemment dans cet article, tout appel de `Method.invoke(..)` redirige, grâce au moteur de réflexion Java, vers la séquence d'appels menant jusqu'à `Runtime.exec()` (appels de 17 jusqu'à 23). Maintenant, tout compte fait, la méthode native `invoke0` redirige l'appel vers `Runtime.exec(..)` qui va exécuter la commande pour lancer la calculatrice.

4. RÉPARATION DE LA VULNÉRABILITÉ

La vulnérabilité a été corrigée dans la version 3.2.2 de la bibliothèque Commons Collections. La solution consiste à rajouter une vérification sur l'objet `InvokerTransformer` lors de l'appel de sa méthode `transform()` (ligne 16 de la pile d'appels). Une méthode `checkUnsafeSerialization` d'une classe `FunctorUtils` vérifie si la sérialisation est activée pour des classes non sécurisées comme le `Transformer`. À défaut, une exception est levée.

CONCLUSION

La vulnérabilité CommonsCollections1 exploite les transformateurs qui peuvent être intégralement contrôlés par un attaquant. C'est un danger potentiel pour les applications critiques développées en Java et utilisant la bibliothèque Commons Collections 3.1. Cette vulnérabilité persiste également dans la version 3.2 et n'est corrigée qu'à partir de la version 3.2.2. Les vulnérabilités de désérialisation sont encore découvertes de temps en temps **[8]**. ■

RÉFÉRENCES

[1] Alexandre Bartel, Jacques Klein et Yves Le Traon. Désérialisation Java : une brève introduction. Multi-System & Internet Security Cookbook (*MISC*), 100:73-76, 2018 : **https://connect.ed-diamond.com/MISC/MISC-100/Deserialisation-Java-une-breve-introduction**

[2] Alexandre Bartel, Jacques Klein et Yves Le Traon. Désérialisation Java : une brève introduction au ROP de haut niveau. Multi-System & Internet Security Cookbook (*MISC*), 101:14-19, 2019 : **https://connect.ed-diamond.com/MISC/MISC-101/Deserialisation-Java-une-breve-introduction-au-ROP-de-haut-niveau**

[3] LazyMap, **https://commons.apache.org/proper/commons-collections/apidocs/org/apache/commons/collections4/map/LazyMap.html**

[4] Ysoserial, **https://github.com/frohoff/ysoserial**

[5] Apache Commons Collections, **https://commons.apache.org/proper/commons-collections/**

[6] PayPal engineering bug, **https://medium.com/paypal-engineering/lessons-learned-from-the-java-deserialization-bug-cb859e9c8d24**

[7] SFMTA bug, **https://arstechnica.com/information-technology/2016/11/san-francisco-transit-ransomware-attacker-likely-used-year-old-java-exploit/**

[8] Cisco Security Manager Java Deserialization Vulnerability, **https://tools.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-20191002-sm-java-deserial**

[9] Proxy, **https://docs.oracle.com/javase/7/docs/api/java/lang/reflect/Proxy.html#Proxy(java.lang.reflect.InvocationHandler)**