Java安全漫谈 - 09.反序列化篇(3)

> 这是代码审计知识星球中Java安全的第九篇文章。

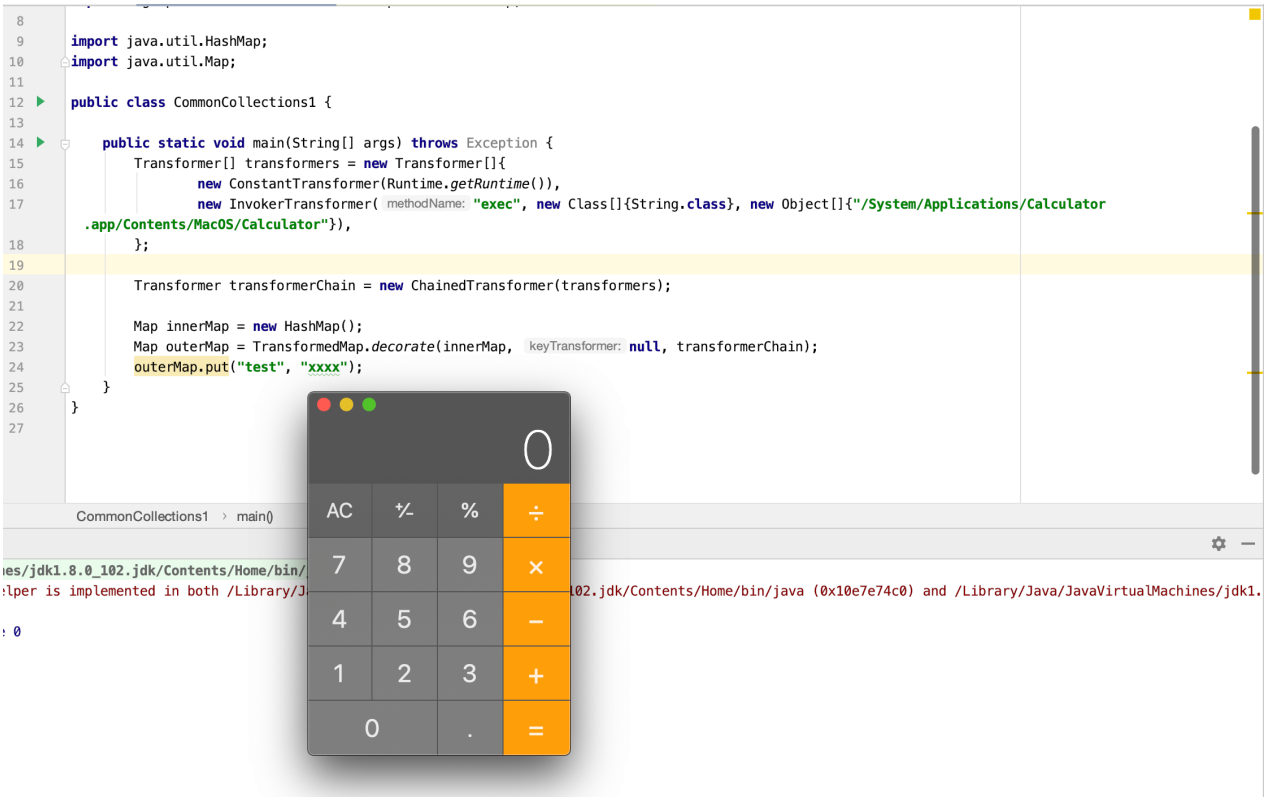
前面给大家介绍了URLDNS这条反序列化利用链，十分简单。但是，逃不过的还是要学，Common-Collections利用链几乎是反序列化学习不可逃过的一关。

网上教程中的demo代码多半比较复杂，干扰要素较多，而且基本都来自于最初几个教程，很多写文章的人只是照着前人教程复现一遍，其实不一定对此有深刻的理解。另外我也不建议直接读ysoserial的源码，因为ysoserial的源码中对代码进行了很多优化，导致理解起来可能比较难一点，这块后面我再详细说明。

虽说确实相比于URLDNS要复杂一些，但我将CommonCollections1利用链简化成了（我独创的哦）如下demo代码（基本已经简化到不能再简化了）：

```
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},
   new Object[]
   {"/System/Applications/Calculator.app/Contents/MacOS/Calculator"}),
17         };
18
19         Transformer transformerChain = new
   ChainedTransformer(transformers);
20
21         Map innerMap = new HashMap();
22         Map outerMap = TransformedMap.decorate(innerMap, null,
   transformerChain);
23         outerMap.put("test", "xxxx");
24     }
25 }
```

将上述代码中的计算器地址`/System/Applications/Calculator.app/Contents/MacOS/Calculator`替换成你本地环境里的计算器路径，运行就会发现弹出了计算器：



这个过程涉及到下面几个接口和类：

TransformedMap

TransformedMap用于对Java标准数据结构Map做一个修饰，被修饰过的Map在添加新的元素时，将可以执行一个回调。我们通过下面这行代码对innerMap进行修饰，传出的outerMap即是修饰后的Map：

```
Map outerMap = TransformedMap.decorate(innerMap, keyTransformer,
valueTransformer);
```

其中，keyTransformer是处理新元素的Key的回调，valueTransformer是处理新元素的value的回调。

我们这里所说的”回调“，并不是传统意义上的一个回调函数，而是一个实现了Transformer接口的类。

Transformer

Transformer是一个接口，它只有一个待实现的方法：

```
public interface Transformer {
    public Object transform(Object input);
}
```

TransformedMap在转换Map的新元素时，就会调用transform方法，这个过程就类似在调用一个”回调函数“，这个回调的参数是原始对象。

ConstantTransformer

ConstantTransformer是实现了Transformer接口的一个类，它的过程就是在构造函数的时候传入一个对象，并在transform方法将这个对象再返回：

```
public ConstantTransformer(Object constantToReturn) {
  super();
  iConstant = constantToReturn;
}

public Object transform(Object input) {
  return iConstant;
}
```

所以他的作用其实就是包装任意一个对象，在执行回调时返回这个对象，进而方便后续操作。

InvokerTransformer

InvokerTransformer是实现了Transformer接口的一个类，这个类可以用来执行任意方法，这也是反序列化能执行任意代码的关键。

在实例化这个InvokerTransformer时，需要传入三个参数，第一个参数是待执行的方法名，第二个参数是这个函数的参数列表的参数类型，第三个参数是传给这个函数的参数列表：

```
public InvokerTransformer(String methodName, Class[] paramTypes, Object[]
args) {
  super();
  iMethodName = methodName;
  iParamTypes = paramTypes;
  iArgs = args;
}
```

后面的回调transform方法，就是执行了input对象的iMethodName方法：

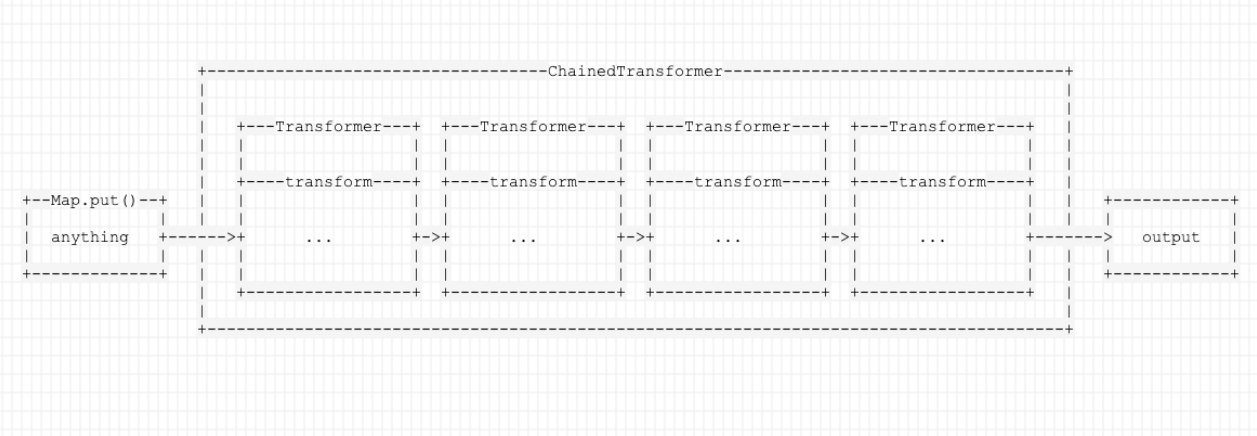
```
public Object transform(Object input) {
  if (input == null) {
    return null;
  }
  try {
    Class cls = input.getClass();
    Method method = cls.getMethod(iMethodName, iParamTypes);
    return method.invoke(input, iArgs);

  } catch (NoSuchMethodException ex) {
    throw new FunctorException("InvokerTransformer: The method '" +
iMethodName + "' on '" + input.getClass() + "' does not exist");
  } catch (IllegalAccessException ex) {
```

```
    throw new FunctorException("InvokerTransformer: The method '" +
iMethodName + "' on '" + input.getClass() + "' cannot be accessed");
  } catch (InvocationTargetException ex) {
    throw new FunctorException("InvokerTransformer: The method '" +
iMethodName + "' on '" + input.getClass() + "' threw an exception", ex);
  }
}
```

ChainedTransformer

ChainedTransformer也是实现了Transformer接口的一个类，它的作用是将内部的多个Transformer串在一起。通俗来说就是，前一个回调返回的结果，作为后一个回调的参数传入，我们画一个图做示意：



它的代码也比较简单：

```
public ChainedTransformer(Transformer[] transformers) {
  super();
  iTransformers = transformers;
}

public Object transform(Object object) {
  for (int i = 0; i < iTransformers.length; i++) {
    object = iTransformers[i].transform(object);
  }
  return object;
}
```

理解demo

了解了这几个Transformer的意义以后，我们再回头看看demo的代码。

这两段代码就比较好理解了：

```
Transformer[] transformers = new Transformer[]{
                new ConstantTransformer(Runtime.getRuntime()),
                new InvokerTransformer("exec", new Class[]{String.class},
new Object[]
{"/System/Applications/Calculator.app/Contents/MacOS/Calculator"}),
};
Transformer transformerChain = new ChainedTransformer(transformers);
```

我创建了一个ChainedTransformer，其中包含两个Transformer：第一个是ConstantTransformer，直接返回当前环境的Runtime对象；第二个是InvokerTransformer，执行Runtime对象的exec方法，参数是`/System/Applications/Calculator.app/Contents/MacOS/Calculator`。

当然，这个transformerChain只是一系列回调，我们需要用其来包装innerMap，使用的前面说到的`TransformedMap.decorate`：

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

最后，怎么触发回调呢？就是向Map中放入一个新的元素：

```
outerMap.put("test", "xxxx");
```

如何生成一个可利用的序列化对象?

当然，上面的代码执行demo，它只是一个用来在本地测试的类。在实际反序列化漏洞中，我们需要将上面最终生成的outerMap对象变成一个序列化流。

我们如何生成一个可以利用的反序列化POC呢？中间又会遇到哪些问题呢?

这个问题留在下次再给大家说说。

这一篇我们用一个不能够再简化的demo给大家演示了CommonCollections利用链的执行流程，而且我特地去掉了和反射相关的代码，所以大家可以看到本节demo中并没有涉及到反射相关的内容，也可以让大家尽量不要被其他和核心知识不相关的内容干扰理解。