有思考的 Java 安全生态

XStream

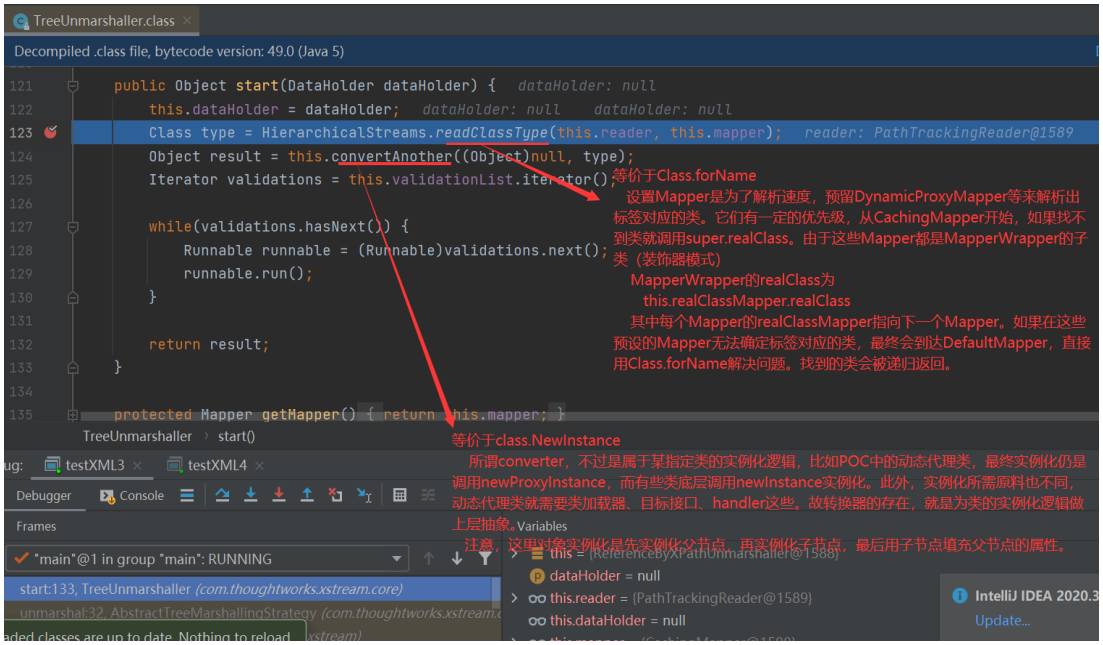
参考博客:（1）Java 安全之 XStream 漏洞分析
https://www.cnblogs.com/nice0e3/p/15046895.html
（2）SpringMVC XStreamMarshaller 反序列化漏洞剖析
https://xz.aliyun.com/t/2602
（3）CVE-2020-26217/26259 Xstream 远程代码执行/任意文件删除漏洞分析
https://xz.aliyun.com/t/8694
（4）XStream 官网
https://x-stream.github.io/security.html
（5）XStream 源码解析
https://www.jianshu.com/p/387c568faf62

前面我想到 tomcat 内存马之外的 3 条线，weblogic t3、fastjson 和 XStream。Weblogic t3 本身就是 rmi 的实现，对它的攻击方式有二阶反序列化绕过对 CC 链等恶意类的黑名单、JRMP 远程带外一阶、以及寻找 CC 链的替代品。Fastjson 是做 JSON 和 Java 对象的转换，对它的攻击方式是 templatesImpl 直接命令执行、以及 jdbcrowsImpl 链实现 JNDI 注入。XStream 是做 XML 和 Java 对象的转换，目前浏览的对它的攻击方式是直接命令执行。从常见的利用上看，weblogic t3 和 fastjson 喜欢 rmi、JNDI 那一套远程调用；从漏洞发生环境来看，fastjson 和 XStream 都是在解析这种标签文档时触发，和 weblogic t3 的 RMI 这种基础设施背景有稍许不同。我们也可以清晰地看到直接命令执行、JNDI 注入等方式在逐渐变成底层工具，我们的工作抽象在中间件的运行机制上，底层工具都有机会利用。但是，我看到几位师傅做 weblogic t3 时没有补丁都是读原理，fastjson 可能会被时代抛弃，觉得还是 XStream 潜在收益高些。还是以 XStream 为主，学完 nice_0e3 师傅到 1.4.10 版本的漏洞博客，再从官方文档和其他资料上追到最新的漏洞。

0x01 XStream 运行机制

XStream 是用来将 Java 对象和 XML、JSON 相互转化的框架。Xstream 序列化 XML 时需要引用的 jar 包：xstream-[version].jar、xpp3-[version].jar（本环境没用到）、xmlpull-[version].jar。Xstream 序列化 Json 需要引用的 jar 包：jettison-[version].jar。

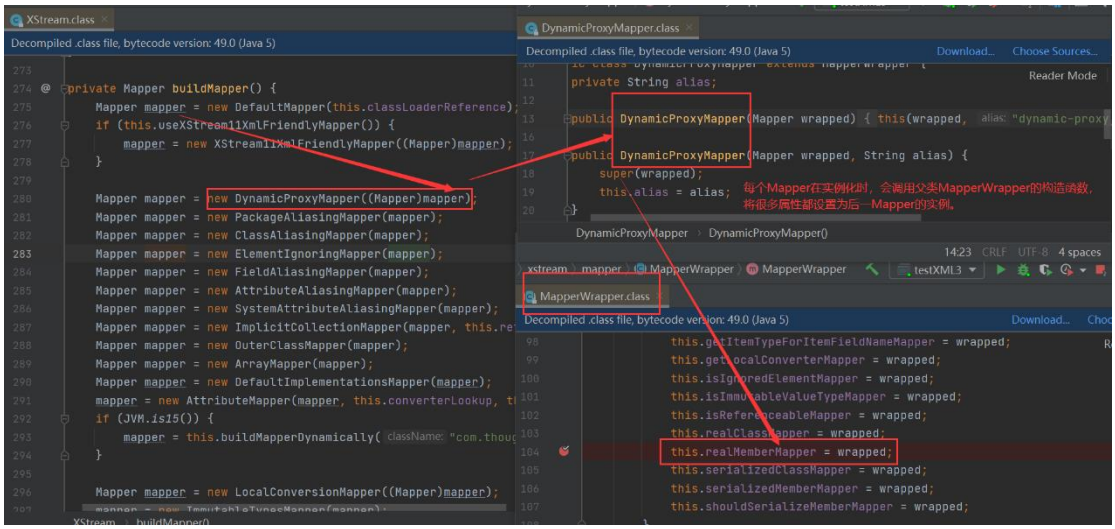
读完几位师傅的博客后，我觉得几位师傅的分析很细节，但我仍然无法将思路连贯起来，我需要真正的锚点。又经历几小时的调试分析后，我将 XStream 解析 XML 的宏观思路放在下面这张图，图中的两段话概括了 XStream 查找类和实例化类的过程，虽有丢失细节之嫌：



上图 realClassType 是根据标签名获取 Class 对象。当某 Mapper 不能获取到 Class 对象时，就会调用父类 MapperWrapper 的 realClass 方法：

```
public Class realClass(String elementName) { return this.realClassMapper.realClass(elementName); }
```

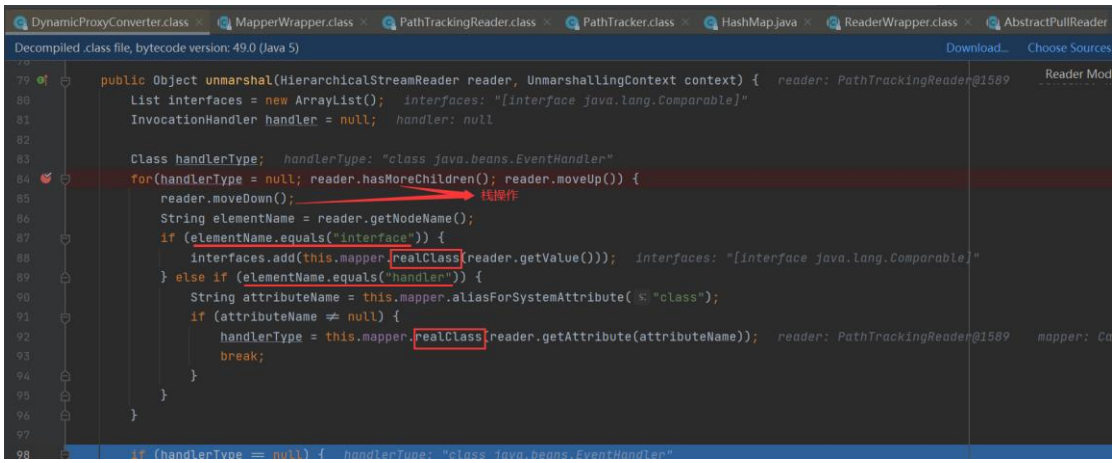
比如 sorted-set 标签解析后 elementName 为 java.util.SortedSet，DynamicProxyMapper 无法获取其 Class 对象，就会调用自身 realClassMapper 属性的 realClass 方法。通过下图可知 XStream 类实例化时构建了这些 Mapper，以 DynamicProxyMapper 实例化过程为例，其构造函数的参数 wrapped 为 DefaultMapper 实例， 进一步可知 DynamicProxyMapper 的众多属性都指向 DefaultMapper 实例。包括这里关键的 realClassMapper 属性。



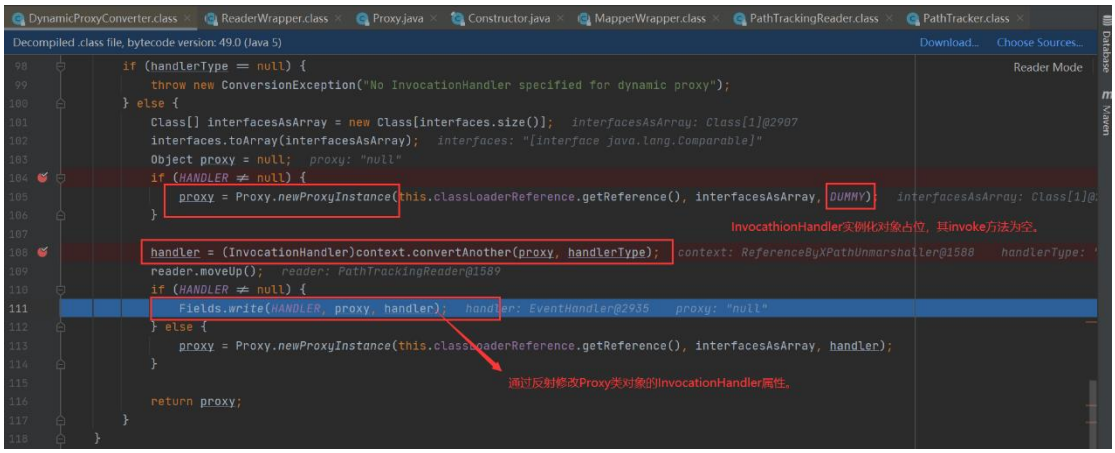
这里父类 MapperWrapper 是装饰者，包装了不同的映射器，递归确定 Class 对象的过程都是通过 super.realClass 方法。这里提一下师傅写的优先级关系(可以直接在 buildMapper 方法中看到)，DefaultMapper 作为王牌优先级最低，所以，在我看来，这一条优先级链就是为了提高 XStream 解析速度而设计的（**经典思想：加速大概率事件**)：

CachingMapper-->SecurityMapper-->ArrayMapper--->ClassAliasingMapper-->PackageAliasingMapper--->DynamicProxyMapper--->DefaultMapper

接下来以 DynamicProxyConverter$**unmarshal** 方法为例讲讲转换器的关键逻辑。图中标注皆为运行后的值，并非初始值。首先是通过 reader 获取子标签（子节点），将子标签与动态代理实例化时所需材料比较后，再通过 realClass 方法获取到对应的 Class 对象。这里涉及到大量堆栈操作，比如在处理一个标签时，就会调用 reader.moveDown 方法弹栈，维护栈结构使得能顺利处理下一个标签。



NewProxyInstance 实例化动态代理时，需要类加载器、目标接口、以及 InvocationHandler 实例。前面已经获取的 handlerType 为 java.beans.EventHandler 类，传入 convertAnother 方法，以获得 EventHandler 实例。最后，是通过反射将动态代理实例的 InvocationHandler 属性指向 EventHandler 实例。



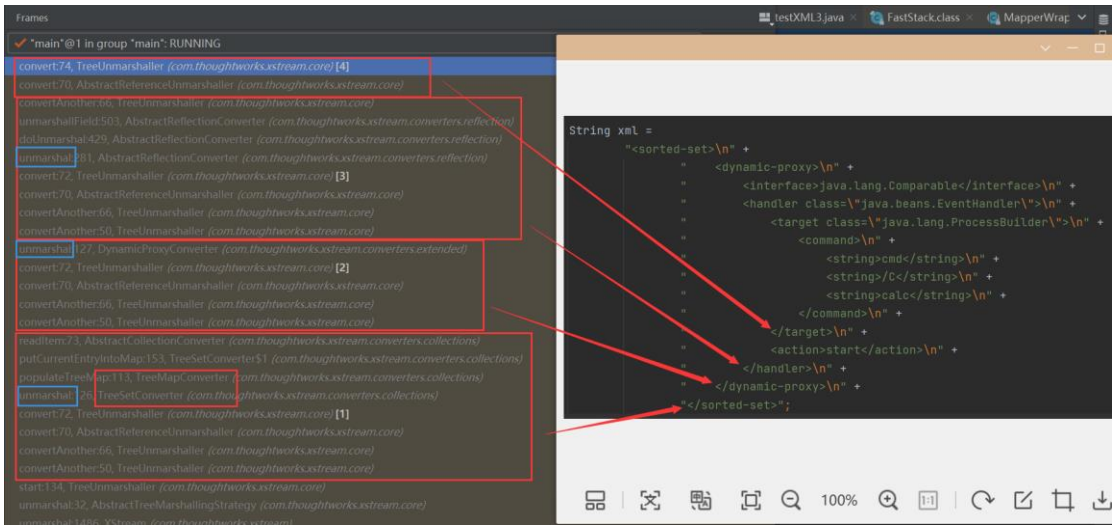
经过调试不难发现转换器的执行逻辑，比如下图，每个转换器在**调用其 unmarshal 方法进行解组前**，都经过这样的调用链：

TreeUnmarshaller$convertAnother1→TreeUnmarshaller$convertAnother2→AbstractReferenceUnmarshaller$covert→TreeUnmarshaller$convert→转换器$unmarshal

这种调用链实际是策略模式，即根据不同的场景创建不同的 TreeUnmarshaller 子类解决问题。AbstractReferenceUnmarshaller 类的父类就是 TreeUnmarshaller 类，其子类是 ReferenceByXPathUnmarshaller 类，调试发现链中的 Unmarshaller 实例都是同一个（XStream 默认的）ReferenceByXPathUnmarshaller 实例。

调用链 convertAnother2 方法会查找类对象对应的转换器，而 TreeUnmarshaller$convert 方法会把 TreeUnmarshaller 类的 reader 属性传递给该转换器的解组逻辑 unmarshal 方法，我们知道，reader 属性记录了 XML、栈结构及其位置等。

下图也表明了解析 CVE-2013-7285 的 POC 时 convert 函数栈的递归调用。



至此，XStream 解析 XML 的运行机制已经理清，下面来看看漏洞原理。

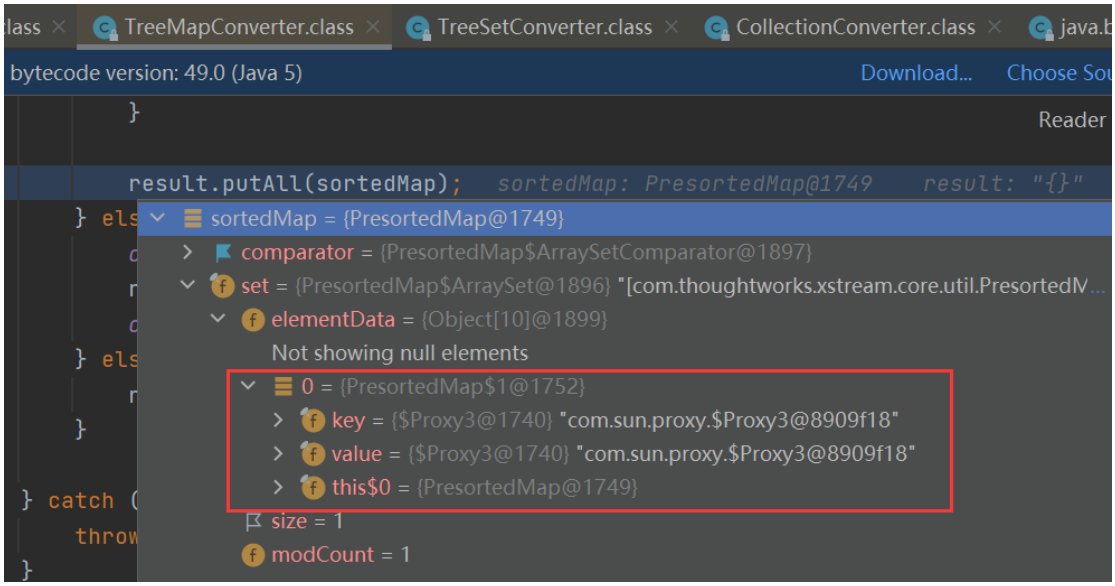
0x02 CVE-2013-7285

复现环境：XStream1.4.10+JDK7

官方 POC：https://x-stream.github.io/CVE-2013-7285.html

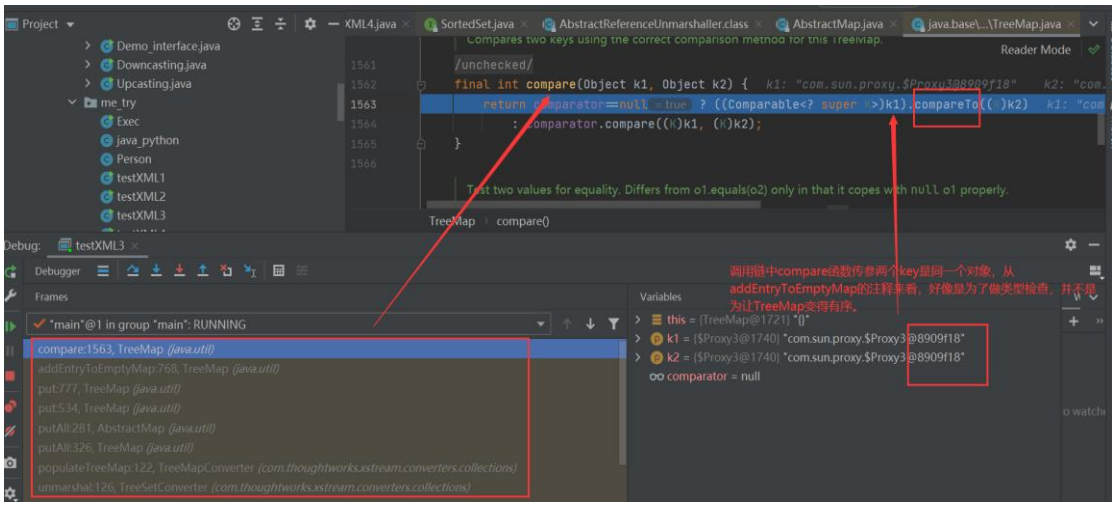
https://www.cnblogs.com/nice0e3/p/15046895.html#0x03-%E6%BC%8F%E6%B4%9E%E5%88%86%E6%9E%90

官方 POC 外层节点为 sortedset 接口，其实也可修改为 treemap 类，漏洞触发原理相同。观察上图小红框部分就能明白这点，在解组 sorted-set 节点时，使用了 TreeSetConverter 的 unmarshal 方法，该方法会调用 TreeMapConverter 的 populateTreeMap 方法，而 TreeMapConverter 的 unmarshal 方法也会调用该方法。



漏洞就发生在 populateTreeMap 方法中，**它在 XStream 解组所有标签后递归返回时，会将实例化对象放入 TreeMap 实例。**TreeMap 的 addEntryToEmptyMap 方法会调用 compare（key，key）做类型检查（防 Null），其实现如下图。这里的 k1 是解组得到的动态代理实例，该动态代理实例实现的是 Comparable 接口，调用该接口的 compareTo 方法，就会触发动态代理绑定的 EventHandler$invokeInternal 的 MethodUtil.**invoke** 方法。由于 XML 可控，我

们将 EventHandler 的 target 属性已经设置为 ProcessBuilder 实例，action 属性置为了“start”，这样调用 MethodUtil.invoke 最终执行 ProcessBuilder$start 方法。当然可以为所欲为。



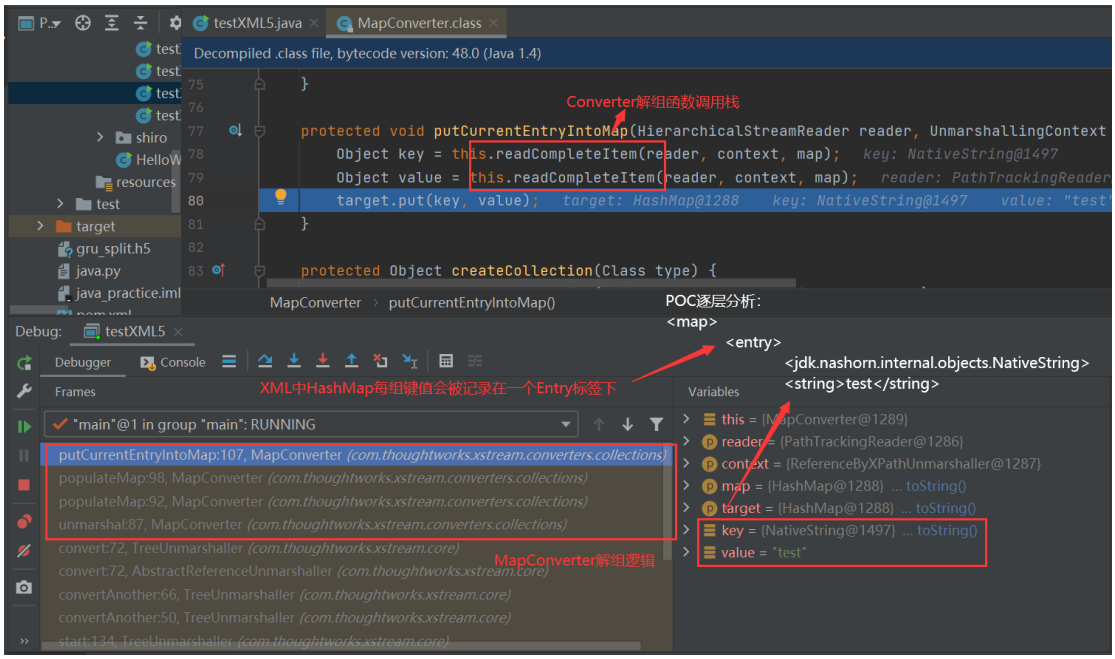
总结：XStream 解析 sorted-set 标签时用到 TreeMapConverter 转换器，其运行逻辑中对 Comparable 实例调用 compareTo 方法，能被构造的动态代理实例利用，造成命令执行。也就是说，对于 Xstream 解组时函数调用链，如果运行逻辑中某函数调用可控变量的方法，那么就可以将可控变量构造为动态代理实例，进一步通过 handler 达到命令执行。

0x03 CVE-2020-26217

复现环境：XStream1.4.13+JDK8

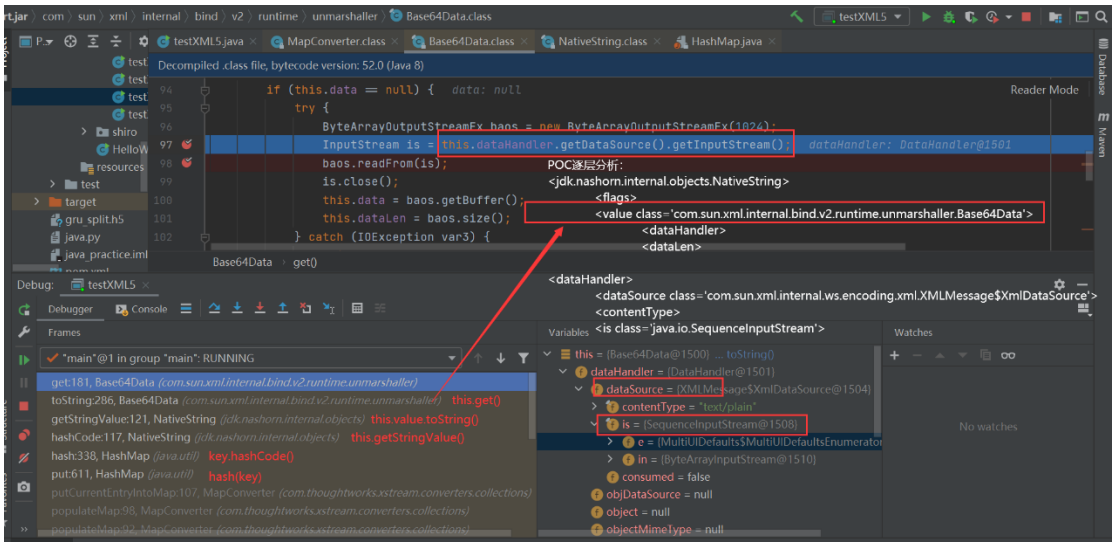
官方 POC: https://x-stream.github.io/CVE-2020-26217.html

XStream 解组 payload 外层 map 标签时，从 TreeUnmarshaller$start 进入 MapConverter 转换器。由下图可以看到该转换器的解组逻辑，在转换器的 putCurrentEntryIntoMap 方法中会继续 converter 解组函数栈，解组结束后会用 HashMap.put 储存实例。

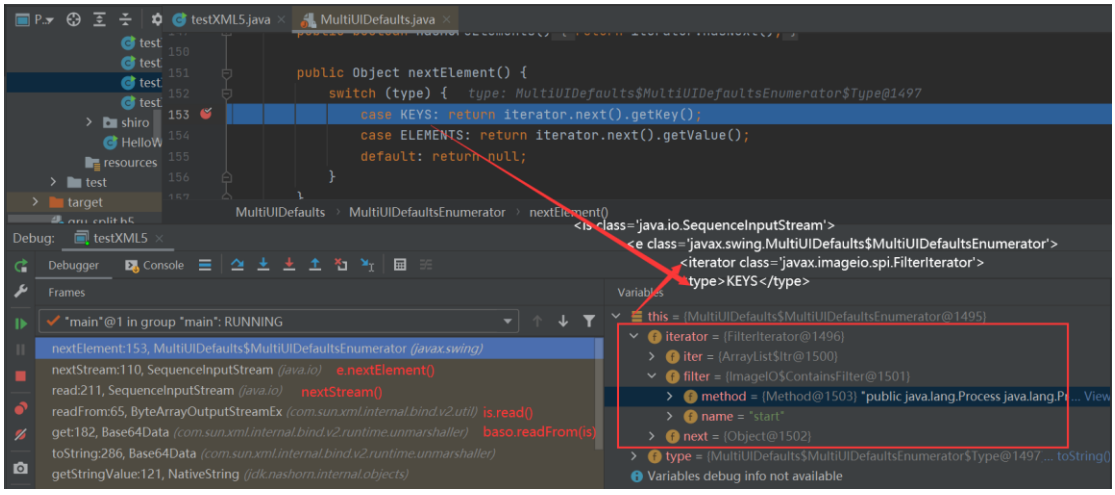


结合 CVE-2013-7285 的调试经验可知，解组过程只是为设置变量，解组完毕储存在 map、set 时才会在调用链上触发漏洞。这个漏洞就发生在 HashMap.put 的调用链上。接下来是一

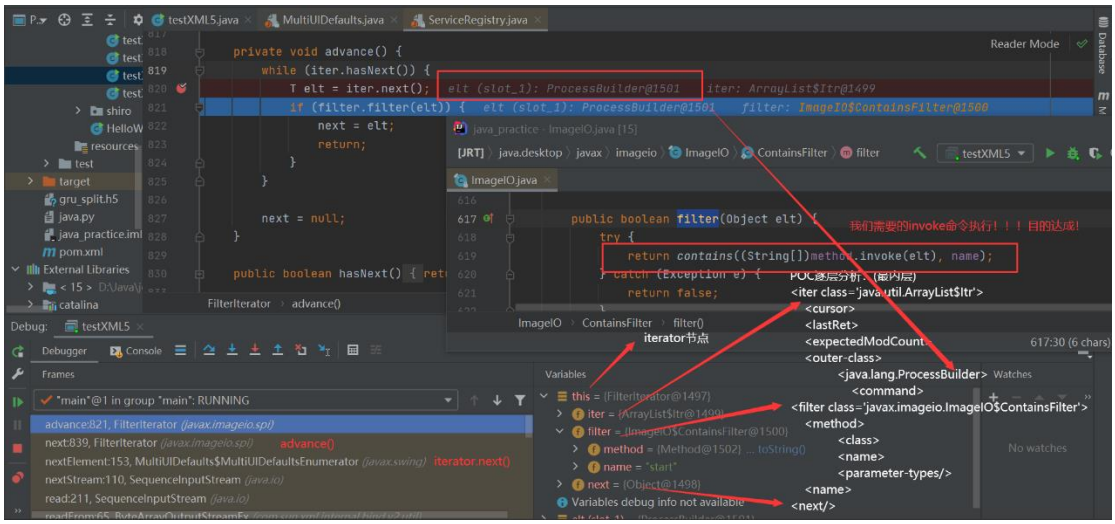
组实例的获取过程，HashMap→NativeString 实例→Base64Data 实例→DataHandler 实例→XmlDataSource 实例→SequenceInputStream 实例。解组过程已经对这些类实例化，从将 NativeString 实例放入 HashMap 存储开始，通过运行逻辑中对实例属性的获取，我们逐步走向 invoke 命令执行点。下图反映了此过程和 payload 标签的关系，获取到的 is 变量为 SequenceInputStream 实例，接下来分析 baos.readFrom(is)。



继续调试，调用链进入 baos.readFrom(is)，is 的 e 属性为 MultiUIDefaultsEnumerator 实例，跟踪进入 e 的 nextElement 方法如下图。XML 中 e 节点下的 type 节点、iterator 节点分别是 KEYS 和 FilterIterator 类，在 converter 递归解组时已经实例化，因此 switch 判断后会进入 iterator.next()。



iterator.next()，也就是 FilterIterator 实例的 next 方法，继续调试就会到达 POC 最关键的部分，如下图 advance 方法。已知 POC 中构造好了 FilterIterator 实例的 iter、filter 和 next 属性。advance 中执行 iter.next()，会将 POC 中构造的 ProcessBuilder 实例赋值给 elt。再调用 ContainsFilter 实例#filter 方法，跟踪进入该方法，可以看到其执行了 method.invoke(elt)，**这里 method 是 ProcessBuilder 类的 start 方法，elt 是 ProcessBuilder 实例**，通过反射实现了命令执行。



我觉得漏洞挖掘的经验就是代码块的拼接，就像专业不同的马仔，有的以“某函数为标志”就能承接运行逻辑，有的能“从某函数开始必达命令执行”。比如，这个 POC 的 Base64Data$toString 以及 SequenceInputStream$read 都是走向 method.invoke 的重要一环，其他位置大多是顺水推舟之事（属性构造）。此外，漏洞调试真点点点，目前如果把 advance 方法的断点取消掉，iterator.next()内部不论是设置断点，还是单步调试都进不去，直接弹 calc。能写出这部分漏洞分析，很多时候都是自己查找前面师傅分析的类、设置断点，才**有可能**成功截停。以前调试 CC 链时也遇到这样的情况，取消勾选 Enable “toString” Object view 解决了部分问题，或许后面会突然明白吧。

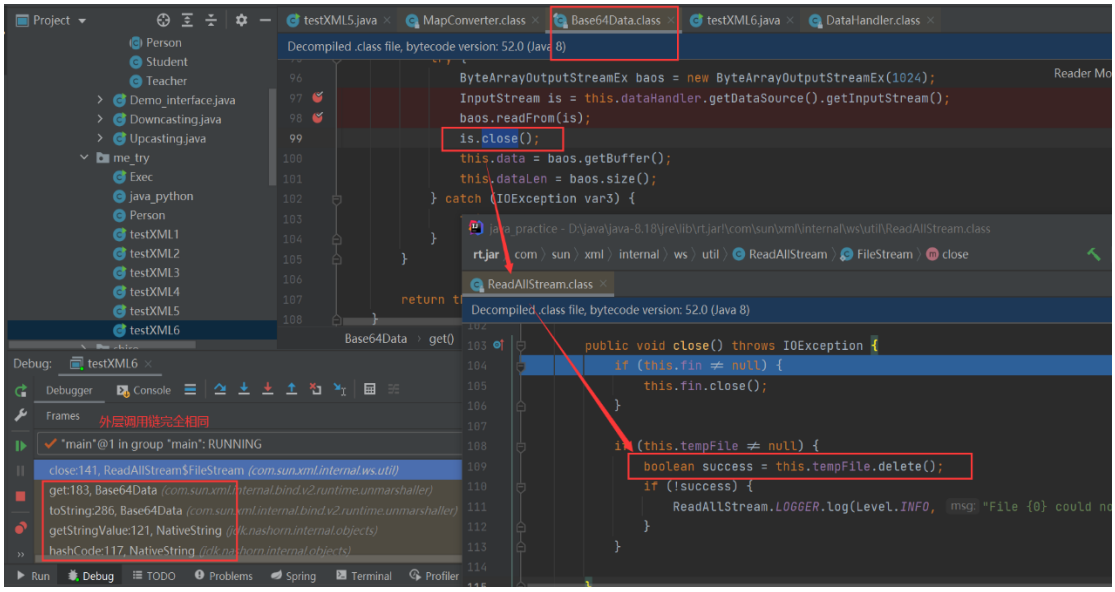
0x03 CVE-2020-26259

复现环境：XStream1.4.13+JDK8

官方 POC: https://x-stream.github.io/CVE-2020-26259.html

```
String xml = "<map>\n" +
        "  <entry>\n" +
        "    <jdk.nashorn.internal.objects.NativeString>\n" +
        "      <flags>0</flags>\n" +
        "      <value class='com.sun.xml.internal.bind.v2.runtime.unmarshaller.Base64Data'>\n" +
        "        <dataHandler>\n" +
        "          <dataSource class='com.sun.xml.internal.ws.encoding.xml.XMLMessage$XmlDataSource'>\n" +
        "            <contentType>text/plain</contentType>\n" +
        "            <is class='com.sun.xml.internal.ws.util.ReadAllStream$FileStream'>\n" +
        "              <tempFile>D:\\tmp_error_SearchRegustByBFS.txt</tempFile>\n" +
        "            </is>\n" +
        "          </dataSource>\n" +
        "          <transferFlavors/>\n" +
        "        </dataHandler>\n" +
        "        <dataLen>0</dataLen>\n" +
        "      </value>\n" +
        "    </jdk.nashorn.internal.objects.NativeString>\n" +
        "    <string>test</string>\n" +
        "  </entry>\n" +
        "</map>";
```


观察 POC 和下图调用栈可知，该漏洞的外层调用链与 CVE-2020-26217 完全相同，只是这里 XMLDataSource 实例的 is 属性，指向 ReadAllStream$FileStrem 实例，而不是前面的 SequenceInputStream 实例。漏洞发生点不再是 baos.readFrom(is)，而是 is.close()。在 FileStrem 实例的 close 方法中，会对 FileStrem 实例的 tempFile 属性所指向的文件路径执行 delete()。tempFile 在 XML 中是可控变量，所以该漏洞能达到任意文件删除的目的。



熊本熊本熊师傅的总结是，只要找到一个类，其 close 方法有能被利用的地方，就算挖到了一条新的利用链。本漏洞是，FileStrem 类的 close 方法能提供文件删除。所以，所谓经验应该就是要对一些关键函数的敏感程度，这些函数承上启下，背后站着的一个个类构成了我们的武器库。

0x04 学习小结

1. 越发觉 java 安全最重要的是对中间件运行机制的理解。掌握中间件的运行机制，比如 tomcat 的调用链、XStream、FastJson 的解析过程等可能最开始需要大量知识铺垫，但是，后续不管是开发还是调试的经验积累，就有可能突然明白某些类是否可被利用、如何配合运行机制组装类以达成恶意目的。

2.读博客的时候，对某个 CVE 我们会集中精力去理解它的调用链，也就是它自己的漏洞原理，比如 XStream 的 CVE-2013-7285，我们总结可能就是一句，“对动态代理类调用时会触发 handler$invoke”。但是，这种理解是别人博客最想表达的内容，局限在一个洞上，而不是运行机制，不是关键点。应该这样去说，“TreeMapConverter 将 XML 解析结果最终放入 TreeMap 储存时，会调用 compare 进行比较，底层调用的 compareTo 触发了精心构造的动态代理类”。大概就是这种将重心转移在中间件的意思，这种理解需要调试工作来支撑。

3.调试前要先睡觉，另外要把 setting->Debuger->DataViews->Java->Enable “toString” Object view 取消勾选。

4.本来打算追问 XStream 漏洞，但发现新的 POC 大多涉及 JNDI、RMI、LDAP 等内容，看来命令执行很有可能快被黑白名单封死了。所以还是继续 fastjson、weblogic T3 两条线吧，期间掌握好 JNDI 这一系列的方法。此外，越来越觉得不管是 JAVA 序列化、fastjson 还是 XStream，它们的解析逻辑本质上快相通了，只是涉及的代码逻辑不同，或许以后这种感觉会清晰起来。