有思考的 Java 安全生态

Weblogic T3

参考博客：

（1）.weblogic 历史 T3 反序列化漏洞及补丁梳理
http://redteam.today/2020/03/25/weblogic%E5%8E%86%E5%8F%B2T3%E5%8F%8D%E5%BA%8F%E5%88%97%E5%8C%96%E6%BC%8F%E6%B4%9E%E5%8F%8A%E8%A1%A5%E4%B8%81%E6%A2%B3%E7%90%86/

（2）.从 Weblogic 原理上探究 CVE-2015-4852、CVE-2016-0638、CVE-2016-3510 究竟怎么一回事
https://xz.aliyun.com/t/8443

（3）.Java 安全之原生 readObject 方法解读
https://www.cnblogs.com/nice0e3/p/14127885.html

（4）.WebLogic 安全研究报告
https://mp.weixin.qq.com/s?__biz=MzU5NDgxODU1MQ==&mid=2247485058&idx=1&sn=d22b310acf703a32d938a7087c8e8704

（5）. .CVE-2017-3248——WebLogic 反序列化初探
https://www.anquanke.com/post/id/225137#h3-3·

（6）. weblogic Coherence 组件漏洞总结分析
https://xz.aliyun.com/t/10016

（7）. CVE-2020-2555——Coherence 反序列化初探
https://www.anquanke.com/post/id/226270

（8）. CVE-2020-2883——WebLogic 反序列化初探
https://www.anquanke.com/post/id/227604

（9）. CVE-2020-14645——WebLogic 反序列化
https://www.anquanke.com/post/id/231425

（10）. WebLogic CVE-2021-2394 RCE 漏洞分析
https://blog.riskivy.com/weblogic-cve-2021-2394-rce%e6%bc%8f%e6%b4%9e%e5%88%86%e6%9e%90/

（11）. How Did I Find Weblogic T3 RCE
https://xz.aliyun.com/t/9068

（12）. weblogic CVE-2021-2394 分析
https://xz.aliyun.com/t/10052

（13）. CVE-2021-2135：Weblogic 二次序列化漏洞分析
https://mp.weixin.qq.com/s/eyZfAPivCkMbNCfukngpzg

结束对几位师傅分析 fastjson 漏洞历史的回溯后，我开始学习月初觉得最难的 Weblogic T3。Weblogic 漏洞在 CNVD 上的报告数目，远超 XStream、fastjson 甚至是 tomcat，且 Oracle 其他产品的漏洞报告经常涉及该容器。目前觉得其反序列化问题大头在 T3、小头在 XmlDecoder，对它们进行研究是比较有价值的。假期最初的想法，是搞清楚 Java 安全生态，也就是明白各组件在开发体系中的位置和安全价值，但是，到了单独追一个组件的漏洞历史的时候，难免因为兴奋，搁置阅读博客而关注最新 POC，这样的效率目前来说还是比较低的。所以，写这篇笔记时，我决定暂时放弃了对 XStream 漏洞的自调计划，还是要多插眼，把战争迷雾给清理好。**因为没有补丁，故这次笔记以理解几位师傅的博客为主赶赶路，**

即非常有照虎画猫之嫌。对于 T3 协议的利用方式，我觉得二阶反序列化重点在原生 readObject 分析，带外 RCE 重点在 JRMP 注入 client 的底层实现分析，最近的漏洞趋势重点则在对 ValueExtractor#extract 和 ExternalizableLite#readExternal 方法的理解。

0x00 原生 readObject

分析 CC 链时，我们常常以目标类的 readObject 方法直接领域展开 Gadget，但实际从组件的反序列化点走到目标类还经过一段标准的调用链，即 ObjectInputStream 的 readObject 调用链，简称原生 readObeject。这段调用链负责的是，从字节流中获取目标类的 Class 对象，实例化目标类后，调用类对象的反序列化方法继续解析。也就是说，不管序列化流被装载在 ByteArrayInputStream 还是 FileInputStream 等 InputStream 子类，都可以且需要被再封装到 ObjectInputStream 才能反序列化。

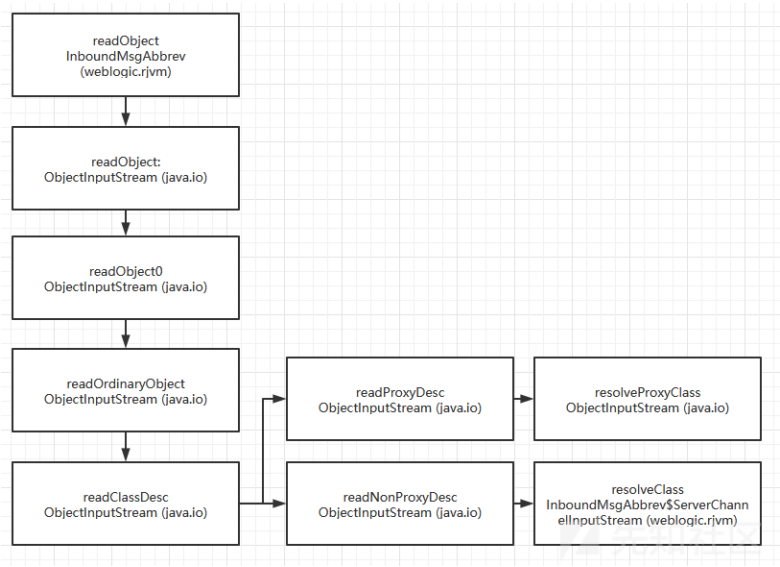
Weblogic 的反序列入口在 InboundMsgAbbrev 的 readObject 方法，该方法调用内部类 ServerChannelInputStream 的 readObject 方法，该内部类同时是 ObjectInputStream 的子类，且没有重写 readObject。因此，Weblogic 反序列化时通过原生 readObject 获取序列化描述符，调用栈执行到 ServerChannelInputStream#resolveClass 获取 Class 对象（多态的体现，父类调用 resolveClass 时，由于引用指向 ServerChannelInputStream，且该子类有 resolveClass 方法，故会调用子类的方法）。ServerChannelInputStream#resolveClass 实际还是通过父类 resolveClass 方法内部调用 Class.forName 获取 Class 对象，但由于 ServerChannelInputStream 的 resolveClass 方法是 Weblogic 自身的反序列化关键位置，我们不可能修改原生 readObject，但可以在这里设置黑名单来过滤恶意类。摘两张“熊本熊本熊”师傅博客中的图，可以看到从序列化描述符中获取类名，在进入黑名单判断：

更新补丁后

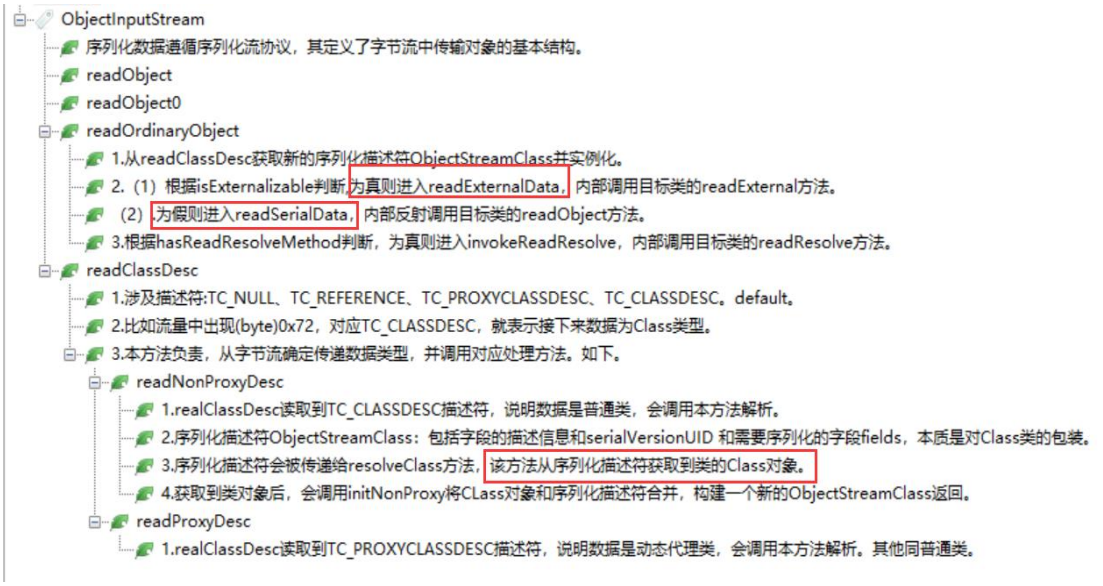
```
protected Class resolveClass(ObjectStreamClass descriptor) throws ClassNotFoundException, IOExcepti
    String className = descriptor.getName();
    if (className != null && className.length() > 0 && ClassFilter.isBlackListed(className)) {
        throw new InvalidClassException("Unauthorized deserialization attempt", descriptor.getName(
    } else {
        Class c = super.resolveClass(descriptor);
        if (c == null) {
            throw new ClassNotFoundException("super.resolveClass returns null.");
        } else {
            ObjectStreamClass localDesc = ObjectStreamClass.lookup(c);
```

更新补丁前

```
protected Class resolveClass(ObjectStreamClass var1) throws ClassNotFoundException,
    Class var2 = super.resolveClass(var1);
    if (var2 == null) {
        throw new ClassNotFoundException("super.resolveClass returns null.");
    } else {
        ObjectStreamClass var3 = ObjectStreamClass.lookup(var2);
```



下面是我对原生 readObject 主线逻辑归纳，从上到下为调用关系。可以看到 readOrdinaryObject 调用 readClassDesc 获取 Class 对象后，会实例化目标类并向下调用其反序列化方法。这就是 CC 链中那些目标类 readObject 方法的调用位置。



0x01 二阶反序列化

CVE-2015-4852 是 T3 漏洞的开端，师傅们的 POC 是使用 CC1 链的 payload 并将之嵌入 T3 数据包，Weblogic 未对反序列化设防故能 RCE。修复方法如上节补丁图，即在 ServerChannelInputStream 的 resolveClass 方法中添加 ClassFilter.isBlackListed 进行黑名单判断，通过后才能 super.resolveClass，从父类 ObjectInputStream 方法中获取 Class 对象。也就是说，它是在 Weblogic 自身的反序列化点上设防，并不会影响到 ObjectInputStream。

Weblogic 反序列化点实际上只是目标类的实例化过程，对目标类内部属性等信息的反序列化，是通过调用目标类自身的 readObject、readExternal、或者 readResolve 实现的。对属性的反序列化肯定还是通过原生 readObject 完成，但不再会经过 Weblogic 自身的反序列化点了，这也就是二阶反序列化漏洞的原因。

CVE-2016-0638，通过 weblogic.jms.common.StreamMessageImpl 的 payload 属性储存

CC1 链的序列化 payload，其 readExternal 如下图（感谢师傅，赶赶路），反序列化过程为：

首先，StreamMessageImpl 能通过 Weblogic 反序列化点 resolveClass 的黑名单，其次，当 readOrdinaryObject 向下调用 StreamMessageImpl 反序列化方法 readExternal 时，内部会将 payload 属性封装在 ObjectInputStream，并展开原生 readObject 调用链。此时，ObjectInputStream 当然不会指向 ServerChannelInputStream，从而不会通过其 resolveClass 方法（中的黑名单），最终能实现恶意类的反序列化。

```
849 @   public void readExternal(ObjectInput var1) throws IOException, ClassNotFoundException {  var1: InboundMsg
850         super.readExternal(var1);
851         byte var2 = var1.readByte();  var2 (slot_2): 1
852         byte var3 = (byte)(var2 & 127);  var3 (slot_3): 1  var2 (slot_2): 1
853         if (var3 >= 1 && var3 <= 3) {
854             switch(var3) {  var3 (slot_3): 1
855             case 1:
856                 this.payload = (PayloadStream)PayloadFactoryImpl.createPayload((InputStream)var1);  var1: Inb
857                 BufferInputStream var4 = this.payload.getInputStream();  var4 (slot_4): BufferInputStreamChun
858                 ObjectInputStream var5 = new ObjectInputStream(var4);  var5 (slot_5): ObjectInputStream@11840
859                 this.setBodyWritable(true);
860                 this.setPropertiesWritable(true);
861
862                 try {
863                     while(true) {
864                         this.writeObject(var5.readObject());  var5 (slot_5): ObjectInputStream@11840
865                     }
```

CVE-2016-0638 的修复，就是在 StreamMessageImpl 类的 readExternal 方法中，将 payload 封装在 FilteringObjectInputStream（继承后者），不再是 ObjectInputStream 类。其 resolve 方法中和 ServerChannelInputStream 一样引入黑名单判断（下图自李三师傅）。由此可知，对于属性反序列化漏洞的修复，都可将 ois 替换为 FilteringObjectInputStream，从而将设防从目标类延展到属性层面。

```
public class FilteringObjectInputStream extends ObjectInputStream {
  public FilteringObjectInputStream(InputStream in) throws IOException {
    super(in);
  }

  protected Class<?> resolveClass(ObjectStreamClass descriptor) throws ClassNotFoundException, IO
    String className = descriptor.getName();
    if (className != null && className.length() > 0 && ClassFilter.isBlackListed(className))
      throw new InvalidClassException("Unauthorized deserialization attempt", descriptor.getName(
    return super.resolveClass(descriptor);
  }
}
```

CVE-2016-3510，同样通过 weblogic.corba.utils.MarshalledObject 绕过 Weblogic 反序列化点的黑名单。MarshalledObject 类的 readResolve 方法，会将属性 objBytes 封装在 ois 并展开原生 readObject。objBytes 属性可以储存恶意序列化 payload，从而实现 RCE。

对 CVE-2016-3510 的修复是重写 ObjectInputStream 的 resolveClass 方法，在其中添加黑名单机制（下图自李三师傅）。

```
  public Object readResolve() throws IOException, ClassNotFoundException, ObjectStreamException {
63  if (this.objBytes == null)
64    return null;
66  ByteArrayInputStream bin = new ByteArrayInputStream(this.objBytes);
67  ObjectInputStream in = new ObjectInputStream(bin) {
      protected Class<?> resolveClass(ObjectStreamClass desc) throws IOException, ClassNotFoundExcep
70      MarshalledObject.filter.check(desc.getName());
71      return super.resolveClass(desc);
      }
    };
74  Object obj = in.readObject();
75  in.close();
76  return obj;
  }
```

二阶反序列化漏洞，我更愿意称之为属性反序列化漏洞，本质是 Weblogic 反序列化点只负责类的反序列化，而对类的属性反序列化则由目标类自己负责，类的属性可控从而对其反序列化实现 RCE。此时，目标类不再是 CC 链中恶意类，而是作为恶意类的装载卡车类。对属性反序列化漏洞的挖掘，装载卡车类需要满足的条件是：具有 byte[]类型的属性，且会将其封装在 ois 中进行反序列化。对其修复，则是在装载卡车类的反序列化方法中添加黑名单机制。从 Weblogic 防御来看，一阶反序列化是在入口点 resolveClass 设防，二阶反序列化是在装在卡车类的反序列化方法设防，偏向于一个点的防御。可能装载卡车类并不好找，所以属性反序列化并没有更多的漏洞。

带外 RCE

T3 的带外 RCE 洞主要通过注入 JRMP client，以及 JNDI 注入实现。对于前者，阅读 northind 师傅的博客时，我发现师傅已经分析到 DGC 层。虽然，我知道理解 RMI 的底层实现的话，会对这套基础工具的使用更加得心应手，但是，目前还是缺少正反馈去做这件事。因此，对于 T3 的带外 RCE 洞，我将根据李三师傅的博客，记录漏洞原理和修复，**不再分析**。

编号	漏洞原理	漏洞修复
CVE-2017-3248	将 JRMPClient 注入服务器，使服务器主动向恶意 JRMPServer 请求恶意类。	ServerChannelInputStream# resolveProxyClass 添加判断，过滤 java.rmi.registry.Registry。
CVE-2018-2628	用 java.rmi.activation.Activator 替换 java.rmi.registry.Registry；用 UnicastRef 也能在反序列化的时候发起 jrmp 请求	UnicastRef 在 weblogic.utils.io.oif. WebLogicFilterConfig 中加进了黑名单。
CVE-2018-2893	streamMessageImpl+jrmp 代理类绕过，streamMessageImpl 的 readExternal 只对普通类拦截，没拦截代理类。UnicastRef 类没被反序列化，但是用到了其信息。	java.rmi.server. RemoteObjectInvocationHandler 在 weblogic.utils.io.oif. WebLogicFilterConfig 中加进了黑名单。
CVE-2018-3245	RemoteObjectInvocationHandler 类被替换，新的类需满足：继承远程类 RemoteObject，没黑。	WebLogicFilterConfig 添加更底层的 java.rmi.server. RemoteObject。
CVE-2018-3191	JtaTransactionManager 类反序列化时，其 initUserTransactionAndTransactionManager 方法能实现 JNDI 注入。	WebLogicFilterConfig 添加 AbstractPlatformTransactionManager。

0x03 新生代：Oracle Coherence 组件安全问题

```
1.CVE-2020-2555
BadAttributeValueExpException#readObject -> LimitFilter.toSting
               ->ChainedExtractor#extract -> ReflectionExtractor#extract

2.CVE-2020-2883
（1）.PriorityQueue#readObject -> readObject -> heapify -> siftDown
         -> siftDownUsingComparator
         -> AbstractExtractor.compare -> MultiExtractor#extract
         -> ChainedExtractor#extract -> ReflectionExtractor#extract
（2）.PriorityQueue#readObject -> readObject -> heapify -> siftDown
         -> siftDownUsingComparator
         -> ExtractorComparator.compare
         -> ChainedExtractor#extract -> ReflectionExtractor#extract

3.CVE-2020-14645
PriorityQueue#readObject -> readObject -> heapify -> siftDown
       -> siftDownUsingComparator
        -> ExtractorComparator.compare
        -> UniversalExtractor#extract -> UniversalExtractor#extractComplex
        -> JdbcRowSetImpl#getDatabaseMetaData -> JdbcRowSetImpl#connect

4.CVE-2020-14756
AttributeHolder#readExternal-> ExternalizableHelper#readObject
    -> ExternalizableHelper#readObjectInternal
    -> ExternalizableHelper#readExternalizableLite
    -> PartialResult#reasdExternal -> PartialResult#add
    -> (AbstractExtractor)MvelExtractor.compare
    -> MvelExtractor#extract -> MVEL.executeExpression

5.CVE-2020-14825
PriorityQueue#readObject -> readObject -> heapify -> siftDown
       -> siftDownUsingComparator
       -> ExtractorComparator.compare
       -> LockVersionExtractor#extract
       -> MethodAttributeAccessor#getAttributeValueFromObject
        -> JdbcRowSetImpl#getDatabaseMetaData -> JdbcRowSetImpl#connect

6.CVE-2020-14841
```

看宽字节安全师傅的博客，发现和 14825 的调用链一样，CVE 官网则提到它通过 IIOP 协议，14825 通过 IIOP、T3 协议，但 14825 涉及的版本要更多。

```
7.CVE-2021-2135
AttributeHolder#readExternal -> ConditionalPutAll#readExternal
    -> ExternalizableHelper#readMap
    -> map.put -> XString#equals -> SimpleBinaryEntry#toString
    -> SimpleBinaryEntry#getKey
    -> ExternalizableHelper#fromBinary -> deserializeInternal -> readObjectInternal
    -> ExternalizableHelper#readExternalizableLite

8.CVE-2021-2394
AttributeHolder#readExternal-> ExternalizableHelper#readObject
    -> ExternalizableHelper#readObjectInternal
    -> ExternalizableHelper#readExternalizableLite
    -> FilterExtractor#readExternal -> FilterExtractor#extract
    -> MethodAttributeAccessor#getAttributeValueFromObject
    -> JdbcRowSetImpl#getDatabaseMetaData -> JdbcRowSetImpl#connect
```

0x04 学习小结

1.docker 命令

```
（1）.docker ps  #查看运行中的容器
  docker ps -a #查看所有容器，包括没运行的
  docker start 容器 id  #启动容器，id 在 ps 里，停止为 stop，删除为 rm
（2）.docker images  #显示镜像，如果以程序文件和进程来理解，镜像相当于程序文件，容器相当于进程
（3）.docker run -d -p 7001:7001 -p 8453:8453 -p 5556:5556 --name weblogic1036jdk7u21 weblogic1036jdk7u21
    #-d 表示后台运行，-p 表示主机端口和容器端口的映射。这是通过镜像启动容器，只在第一次运行时使用。
（4）.docker exec weblogic1036jdk7u21 ls /tmp  在运行的容器中执行命令（只是这台虚拟机的空间）
```

2.将 Weblogic 分片数据包重新组装起来接收:

```python
def recv_basic(the_socket):
    total_data=[]
    while True:
        try:
            data = the_socket.recv(1024)
            if not data:
                break
            total_data.append(data.decode())
        except socket.timeout:
            return ''.join(total_data)
main:
    sock.settimeout(5)
    data = recv_basic(sock)
```

3.Weblogic 就这样匆匆结束了，我跟着师傅们分析了原生 readObject，记录了带外 RCE 的漏洞历史，对于最新的 Coherence 组件安全问题，我根据师傅们的博客总结了调用链。**当然，Coherence 组件是目前小时间段内的趋势，接下来我肯定会花时间自己搭建环境，实际调试这些漏洞，这是我觉得 XStream 之外最想做得事情，感觉有机会追到前沿。**此外，我觉得对反序列化安全问题的学习，花费时间最多的永远都不应该是一个漏洞的原理，而是背景墙上的基础知识，比如原生 readObject、ValueExtractor#extract、ExternalizableLite# readExternalizableLite，慢也就是快了。

4.从 6 月 28 日到 8 月 22 日，我一直坐在电脑前跟着师傅们的博客调试、做笔记，规划并完成对 Tomcat 内存马、XStream、fastjson、Weblogic 四条线的接触，Tomcat 内存马系列我花了整个七月，基本算是总结了 2020 年上半年诸位师傅的爆发性研究。其他线可能在对效率感到小有成就后，就逐渐放缓脚步，至今可能需要重新开始新的征程了。

5.一鼓作气，再而衰，三而竭。虽然，最近几天略显颓唐（也可能是没有 Weblogic 补丁），但是，回想到这两个月。毕业时没有留在西安、父母十分尊重我的时间规划、以及从奶思师傅博客开始地系统性学习，都帮助我妥善利用大部分时间。我觉得，师兄的项目在研究和产品化上已经做得很好了，我不能举棋不定地去帮助师兄，这样只会失去自己对 Java 反序列化坚定信心的最后机会。于是，我厚着脸皮没有再问师兄，我只能羞愧地认为，这是想要做成事而不得不牺牲了。

6.我做“有思考的 Java 安全生态”，虽然题目略显中二，但是，目的是有意义的，我希望能在短时间内确定常见组件在 Java 开发体系的应用位置，并理解其安全价值。现在是接触了 Tomcat 内存马、XStream、fastjson、Weblogic 四条线，我已经发觉组件的安全问题能够单独研究，漏洞依附于组件自身的运行机制，但是，运行机制会有相同点，反序列化链甚至也能相互有所借鉴。此外，反序列化问题与 Web 并不是密不可分的，除 Tomcat 内存马这条线外，我觉得其他线是应用在服务器的其他功能上。**最近看到“云原生”的概念，我也感觉到 Java 安全涉及的组件都是直接影响服务器，所以，解决“云这种基础设施所搭载各种组件的安全问题”，应该也是很有价值的方向，而不必踌躇于 Web 安全。** 这一点需要关注。

7.“完整的工作量，完整的果实”。略显宗教了，但是，希望这是我未来两年坚持的态度。