cve-2019-2729挖掘思路 - cve-2019-2725 bypass

orich1 / 2019-06-22 06:03:00 / 浏览数 8227 安全技术 漏洞分析 顶(1) 踩(0)

前言

Weblogic 当中对 wsdl 中的 soap 消息体解析依赖的是 XMLDecoder 的数据转换

XMLDecoder
其实就是一个将字符串的描述转换成java对象的一个jdk内置工具。因为本身wsdl设计的原因，XMLDecoder的反序列化不用登陆即可利用，处理wsdl消息体就会调用到。

黑名单

Weblogic 中使用XMLDecoder对soap解析的是weblogic.wsee.workarea.WorkContextXmlInputAdapter
这个类，目前已公开使用这个类的包有：bea_wls9_async_response.war 和 wls-wsat.war

每一次的补丁都是从 WorkContextXmlInputAdapter 下手，用黑名单的形式阻止恶意类的创建、恶意函数的调用执行。最新 CVE-2019-2725 patch
限制了标签如下：

```
<object>
<class>
<new>
<method>
<void>
<array>
```

其中 `<object>`■`<class>`■`<new>`■`<method>` 标签是完全禁止出现的，剩下 `<void>`■`<array>` 也有很多限制。

`<void>` 的限制如下：

```
if (qName.equalsIgnoreCase("void")) {
  for (int i = 0; i < attributes.getLength(); i++) {
    if (!"index".equalsIgnoreCase(attributes.getQName(i))) {
      throw new IllegalStateException("Invalid attribute for element void:" + attributes.getQName(i));
    }
  }
}
```

遍历 void 标签的属性，只要存在不是 index 的属性名，就会抛错，无法进行下一步 XMLDecoder 解析

```
if (qName.equalsIgnoreCase("array"))
{
  String attClass = attributes.getValue("class");
  if ((attClass != null) && (!attClass.equalsIgnoreCase("byte"))) {
    throw new IllegalStateException("The value of class attribute is not valid for array element.");
  }
  String lengthString = attributes.getValue("length");
  if (lengthString != null) {
    try
    {
      int length = Integer.valueOf(lengthString).intValue();
      if (length >= WorkContextXmlInputAdapter.MAXARRAYLENGTH) {
        throw new IllegalStateException("Exceed array length limitation");
      }
      this.overallarraylength += length;
      if (this.overallarraylength >= WorkContextXmlInputAdapter.OVERALLMAXARRAYLENGTH) {
        throw new IllegalStateException("Exceed over all array limitation.");
      }
```

如上图，对 array 标签限制也是很严格的，其中可以指定初始化类的 class 属性被检查，如果不是 byte 值，则抛错，对 length 属性也做了大小检查。

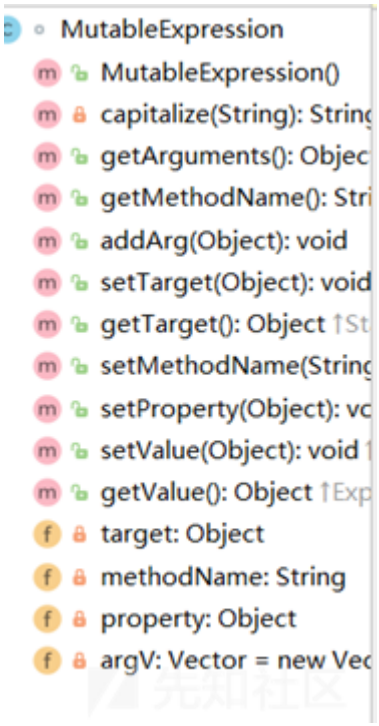
那么除了上述6种标签之外，还有没有标签可以搞事情呢，这就需要深入 XMLDecoder 的解析过程中去了

xml反序列化解析层

XMLDecoder 使用的是 SAXParser 对xml进行解析，其中可以指定 handler，也就是可以在解析各种标签的过程中用 handler 进行某些特殊处理。XMLDecoder 在 JDK6版本中指定的是 ObjectHandler（weblogic10.3.6版本自带的jdk版本）

在ObjectHandler类中主要关注 startElement 和 endElement 函数。解析到某一标签时，startElement函数中会根据当前标签相关信息生成一个 `MutableExpression` 表达式类，在解析到该标签的闭合处时就会执行 endElement 函数，该函数中则会调用 Expression#getValue 计算值，并且根据需要放入父标签的参数列表中

那当下之急就是需要了解 `MutableExpression` 的工作方式，跟进去看看



从结构上看得出还是很简单的，四个属性，其余全是围绕这几个属性进行get、set操作，那么继续深入上文提到的 getValue 函数，跟到了 java.beans.Expression#getValue 中如下：

```
public class Expression extends Statement {
    private static Object unbound = new Object();
    private Object value;

    public Expression(Object target, String methodName, Object[] arguments) {
        super(target, methodName, arguments);
        this.value = unbound;
    }

    public Expression(Object value, Object target, String methodName, Object[] argum
        this(target, methodName, arguments);
        this.setValue(value);
    }

    public Object getValue() throws Exception {
        if (this.value == unbound) {
            this.setValue(this.invoke());
        }

        return this.value;
    }

    public void setValue(Object value) { this.value = value; }

    String instanceName(Object instance) { return instance == unbound ? "<unbound>"
```

上图里的 getValue 函数首先判断 value 是否已经被改变（MutableExpression初始化Expression使用的是第一个构造函数），因为初始化时 value 已经被指定为 unbound 状态。这里有个小细节，假设后续流程中通过 setValue 的操作给 Expression 的 value 值指定为 Object 的对象，那这个判断还是不通过的，因为new的对象会使用新地址。

那么这里大致知道，只要是第一次执行该Expression的时候，肯定是调用了 invoke 函数的，继续跟进去，跟到了java.beans.Statement#invokeInternal中：

```
private Object invokeInternal() throws Exception {
    Object target = this.getTarget();
    String methodName = this.getMethodName();
    if (target != null && methodName != null) {
        Object[] arguments = this.getArguments();
        if (target == Class.class && methodName.equals("forName")) {
            return ObjectHandler.classForName((String)arguments[0]);
        } else {
            Class[] argClasses = new Class[arguments.length];
```

其实可以从很小一段代码就可以确认，这就是在进行动态反射调用了。刚好 target 就是从 MutableExpression#getTarget 获得，methodName就是从MutableExpression#getMethodName获得。

分析流程到这里已然明了，ObjectHandler#startElement 根据当前标签信息生成一个 MutableExpression 对象，结束标签时则执行他，执行的时候就根据 MutableExpression 对象属性中的 target 和 methodName 来指定了反射调用的类以及所需要调用执行的函数

那么此时回到 ObjectHandler#startElement 函数中，看看有哪些标签是可用的，部分代码如下：

```
if (name == "string") {
   e.setTarget(String.class);
   e.setMethodName("new");
   this.isString = true;
} else if (this.isPrimitive(name)) {
   Class wrapper = typeNameToClass(name);
   e.setTarget(wrapper);
   e.setMethodName("new");
   this.parseCharCode(name, attributes);
} else if (name == "class") {
   e.setTarget(Class.class);
   e.setMethodName("forName");
} else if (name == "null") {
   e.setTarget(Object.class);
   e.setMethodName("getSuperclass");
   e.setValue((Object)null);
} else if (name == "void") {
   if (e.getTarget() == null) {
       e.setTarget(this.eval());
   }
} else if (name == "array") {
   subtypeName = (String)attributes.get("class");
   Class subtype = subtypeName == null ? Object.class : this.classForName2(subtypeName);
   length = (String)attributes.get("length");
   if (length != null) {
       e.setTarget(Array.class);
       e.addArg(subtype);
       e.addArg(new Integer(length));
   } else {
       Class arrayClass = Array.newInstance(subtype, 0).getClass();
       e.setTarget(arrayClass);
   }
} else if (name == "java") {
   e.setValue(this.is);
} else if (name != "object") {
   this.simulateException("Unrecognized opening tag: " + name + " " + this.attrsToString(attrs));
   return;
}
```

如上述代码块，这就是 ObjectHandler#startElement 函数中对标签名的解析过程了，可见只有 String、class、null、void、array、java、object 还有一些基础类型：boolean、byte、char、short、int、long、float、double

Jdk6中没有实现 new 标签的解析，那么此时只剩下基础类型标签和String、null、java这些标签未被限制

但是经过上文的分析，java 、null 标签已经被 setValue 操作过，所以哪怕是后续过程我们能够指定 Object 对象都是不行的，他们不会再次执行，排除。此时就只剩下 String 和基础类型，仔细看看，他们都有 setTarget 和 setMethodName 的操作，这意味着，顶多就新建一个相关类而已，反射目标类不可控、所调用函数不可控，gg

这个时候应该更仔细的看一看 startElement 的处理逻辑

```
        this.chars.setLength(0);
        HashMap attributes = this.getAttributes(attrs);
        MutableExpression e = new MutableExpression();
        String className = (String)attributes.get("class");
        if (className != null) {
            e.setTarget(this.classForName2(className));
        }

        Object property = attributes.get("property");
        String index = (String)attributes.get("index");
        if (index != null) {
            property = new Integer(index);
            e.addArg(property);
        }

        e.setProperty(property);
        String methodName = (String)attributes.get("method");
        if (methodName == null && property == null) {
            methodName = "new";
        }

        e.setMethodName(methodName);
        String length;
        String subtypeName;
        if (name == "string") {
            e.setTarget(String.class);
            e.setMethodName("new");
            this.isString = true;
        } else if (this.isPrimitive(name)) {
```

上图这是在对标签名进行解析前的操作，可以看到提前调用了 setTarget 和 setMethodName ，但是后续解析流程中又会再次调用，则达到了覆盖的目的，比如
`<string class="Test" method="orich1" />`
这样的标签会被解析，但是执行到 string 标签解析的时候，会发生如下调用：

```
if (name == "string") {
   e.setTarget(String.class);
   e.setMethodName("new");
   this.isString = true;
}
```

那么在这里我自己指定的 class 和 methodName 都会在解析string标签时被覆盖掉
这里陷入了死胡同，我们继续往下看：

```
    subtypeName = (String)attributes.get("id");
    if (subtypeName != null) {
        this.environment.put(subtypeName, e);
    }

    String idrefName = (String)attributes.get("idref");
    if (idrefName != null) {
        e.setValue(this.lookup(idrefName));
    }

    length = (String)attributes.get("field");
    if (length != null) {
        e.setValue(this.getFieldValue(e.getTarget(), length));
    }

    this.expStack.add(e);
}
```

上图是完成标签名解析后的最后执行流程，还是在对标签中的属性进行进一步解析，其中 id 和 idref 是对应的，id 对应存入的操作， idref 对应取出的操作，lookup函数如下：

```
public Object lookup(String s) {
    Expression e = (Expression)this.environment.get(s);
    if (e == null) {
        this.simulateException("Unbound variable: " + s);
    }

    return this.getValue(e);
}
```

上图可见取出的操作还顺带执行了一次Expression，那么我们可以利用这个逻辑进行一些值的存取操作。回到 startElement 函数中，最后还判断了一个属性：field

获取 field 对应的字符串以后还调用了 getFieldValue ，返回值作为 Expression 的 value 保存，它这里调用了 Expression#getTarget ，在上文的部分代码中可以看到 java 和 object 标签，是不会被覆盖 target 的，java 标签还没有在黑名单中，跟进 getFieldValue继续分析：

```
private Object getFieldValue(Object target, String fieldName) {
    try {
        Class type = target.getClass();
        if (type == Class.class) {
            type = (Class)target;
        }

        Field f = FieldUtil.getField(type, fieldName);
        return f.get(target);
    } catch (Exception var5) {
        if (this.is != null) {
```

上图简单来说就是对某个对象或者某个Class获取指定的属性值

两个思路

这里停一停仔细想想。
第一个思路：如果使用 java 标签指定 class（也就是指定target）的话，那么意思就是对指定Class进行属性值获取，emmmm只能获取到static修饰的属性，并且由于 FieldUtil#getField 调用的是 Class#getField 获取的 Field 对象，所以还只能获取到 public 修饰的，总的来说就是只能获取 public static 修饰的属性值，这就比较蛋疼了。要我做开发也不会傻不拉几的写个public static 还给初始化一个对象值（诶还真别说，jdk1.7的利用方式就是这样，而且我还真不是开发，所以真不了解他们就真的会给public static赋初值）。

第二个思路：这时候在挖掘过程中我想的是从 public static 修饰的属性里偷一个 Object 对象来，依靠 id 和 idref 存取的特性，对 java 标签的 setValue 进行覆盖，也就是重新给它一个 Object 对象，来满足如下的判断，并且引发 invoke 的执行（图中的unbound就是一个Object对象），那么岂不是就能java标签指定任意class任意method了吗：

```
public Object getValue() throws Exception {
    if (this.value == unbound) {
        this.setValue(this.invoke());
    }
```

所以我还老老实实fuzz了一下public static修饰符的属性，结果当然是关机睡觉

玩了2天后又打开idea仔细想想，咋们静态的Class不行，我可以整一个动态的对象呀，emmm不过..连初始化任意类的 class 标签都被封了，还整个锤子对象，只有 String 和一堆基础类可以用，于是又漫无目的的翻看这几个类中的属性值

突然想到一个毛病就是，哪怕是我拿到了 Object 对象，没卵用，java标签无法执行。因为即使是重新赋值 value 也不会和 unbound 相等，地址不同23333。

rce 两个要素

那么又卡住了，重新梳理了一下rce要素：任意类、任意函数

回想 XMLDecoder 在 weblogic 中引发的问题，首先是没有黑名单造成 rce，然后绕过 object 黑名单造成 rce，最近cve-2019-2725的是利用 class 标签创建任意类，在构造函数中完成利用

那么rce两大要素可以在 XMLDecoder 中拆开利用，如果能造成类似 class 标签的效果，指定了任意类，那么在 methodName 不被覆盖的情况下一定会执行任意类的构造函数（上文中对startElement函数的前半段截图中显示，如果methodName为空，则直接指定为 new，这个在后续的反射调用中代指 newInstance 函数）。

那么重点关注一下 setTarget 的调用，在对标签名解析过程中，调用了 setTarget 函数则是对类做了限制，观察到只有三个标签有点东西，java 和 object 标签没有调用 setTarget但是这俩没法利用，还有一个 void 标签：

```
else if (name == "void") {
   if (e.getTarget() == null) {
       e.setTarget(this.eval());
   }
}
```

Void 标签在补丁当中只能含有 index 属性，所以不能用 class 也就不能指定 target，但是可以用 eval 函数的返回结果作为 target ：

```
private Object eval() {
    return this.getValue(this.lastExp());
}
```

如上图，他是执行父标签的Expression，又陷入死循环，没有起始入口点，无法指定任意类

第三个思路：偏离设计原意制造解析误差

既然rce两要素可以拆开用，那我们分析下如果可以指定任意函数的调用呢？
任意函数调用可以想办法调用到 Class.forName 函数，传入字符串加载指定类，那么就可以达到和 class 标签利用的效果

分析下没有调用 setMethodName 的标签名只有 array 、void、java、object，后三者pass
这个 array 记得好像是有限制的，查看一下补丁：

```
if (qName.equalsIgnoreCase("array"))
{
  String attClass = attributes.getValue("class");
  if ((attClass != null) && (!attClass.equalsIgnoreCase("byte"))) {
    throw new IllegalStateException("The value of class attribute is not valid for array element.");
  }
  String lengthString = attributes.getValue("length");
  if (lengthString != null) {
    try
    {
      int length = Integer.valueOf(lengthString).intValue();
      if (length >= WorkContextXmlInputAdapter.MAXARRAYLENGTH) {
        throw new IllegalStateException("Exceed array length limitation");
      }
      this.overallarraylength += length;
      if (this.overallarraylength >= WorkContextXmlInputAdapter.OVERALLMAXARRAYLENGTH) {
        throw new IllegalStateException("Exceed over all array limitation.");
```

如上图，提取了 array 标签中的 class 属性和 length 属性，但是对应 MethodName 的 method 属性未被检测，欸可以用

但是注意 array 标签的 target 是被覆盖了的，也就是反射类被指定，如下图：

```
} else if (name == "array") {
    subtypeName = (String)attributes.get("class");
    Class subtype = subtypeName == null ? Object.class : this.classForName2(subtypeName);
    length = (String)attributes.get("length");
    if (length != null) {
        e.setTarget(Array.class);
        e.addArg(subtype);
        e.addArg(new Integer(length));
    } else {
        Class arrayClass = Array.newInstance(subtype, length: 0).getClass();
        e.setTarget(arrayClass);
    }
```

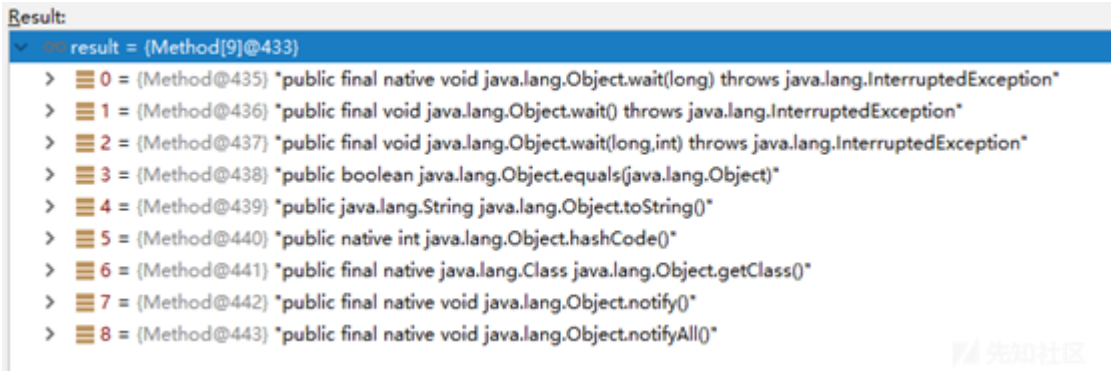
array 标签的 class 只能被指定为 byte ，那么对应的 target 则是 Byte.class ，如果未指定则是 Object.class 。但是不管如何，setTarget 的参数都是 Array 相关，我们没法拿到 Class.class 不能直接执行 forName

虽然目前我们无法指定任意类，但是已经拿到了任意函数。从设计角度出发，array标签是不可能受理 method 属性的（这一点在jdk1.7的xmldecoder设计模型中就处理得很好），利用jdk1.6中代码逻辑问题制造解析误差，达到任意函数调用的效果

又见容错

开始动态调试，用 `<array method="getMethods"` 开始查看有哪些函数

getMethods 只有几个函数，完全不够用，不过有个 getClass 函数，但似乎又没法链式调用，即只能调用一次函数（其实可以通过 id 和 refid 进行链式调用的，但是在这里我就想一发入魂），所以这里也是行不通的



不急，我们还有容错机制帮忙呢，最终反射调用会在java.beans.Statement#invokeInternal里面进行

```
if (target instanceof Class) {
    if (methodName.equals("new")) {
        methodName = "newInstance";
    }

    if (methodName.equals("newInstance") && ((Class)target).isArray()) {
        Object result = Array.newInstance(((Class)target).getComponentType(), arguments.length);

        for(int i = 0; i < arguments.length; ++i) {
            Array.set(result, i, arguments[i]);
        }

        return result;
    }

    if (methodName.equals("newInstance") && arguments.length != 0) {
        if (target == Character.class && arguments.length == 1 && argClasses[0] == String.class) {
            return new Character(((String)arguments[0]).charAt(0));
        }

        m = ReflectionUtils.getConstructor((Class)target, argClasses);
    }

    if (m == null && target != Class.class) {
        m = ReflectionUtils.getMethod((Class)target, methodName, argClasses);
    }

    if (m == null) {
        m = ReflectionUtils.getMethod(Class.class, methodName, argClasses);
    }
} else {
```

上图里，我们的 target 是 Class 的子类实现进入if语句块，不满足后续的3个if，此时 m 没有任何赋值，并且 target 也不是 Class.class
，所以进入上图中的第一个框，该target中对应的 method 就那么几个，如果获取不到会返回 null，于是进入最后一个
if，可以看见为了保证程序执行流程的容错性，它自己给我们指定了获取 Class.class 中的函数，那么就不客气了，直接指定 method="forName"

总结

能造成这个问题主要是因为jdk1.6中xmldecoder的解析代码问题，它将所有的标签属性进行统一处理，但是又没有进行有效性验证，所以精心构造肯定会导致一定的解析偏

还剩下一个可以用的思路，利用 set 和 get 的操作，还有 id 和 refid 属性的特性，构造链式调用

上一篇：CVE-2019-0948：Mic... 下一篇：从0到1掌握反序列化工具之PHPGGC

1. 0 条回复
   - 动动手指，沙发就是你的了！

社区小黑板

目录

RSS 关于社区 友情链接 社区小黑板