title: Fastjson < 1.2.68版本反序列化漏洞分析篇
catalog: true
date: 2020-09-14 14:15:50
subtitle:
header-img:
tags: RCE
categories: Java代码审计

Fastjson < 1.2.68版本反序列化漏洞分析篇

前言

刚接触Java审计，感觉很有趣，希望自己能坚持下去，也希望各位前辈能在看完我的分析后给予专业性的指导。

一、漏洞背景

ClassLoader

ClassLoader称为扩展类加载器，负责加载Java的扩展类库，默认加载JAVA_HOME/jre/lib/ext/目下的所有jar（包括自己手动放进去的jar包）。Java的类加载采用了一种叫做“双亲委托”的方式，所谓“双亲委托”就是当加载一个类的时候会先委托给父类加载器去加载，当父类加载器无法加载的时候再尝试自己去加载，所以整个类的加载是“自上而下”的，如果都没有加载到则抛出 `ClassNotFoundException` 异常。

二、调试分析

期望类

```
    if (!allDigits) {
                                clazz = config.checkAutoType(typeName, null,
    lexer.getFeatures());
                            }
```

当传入的数据不是数字的时候，默认设置期望类为空，进入checkAutoType进行检查传入的类

```
    final boolean expectClassFlag;
```

```
    if (expectClass == null) {
        expectClassFlag = false;
    } else {
        if (expectClass == Object.class
                || expectClass == Serializable.class
                || expectClass == Cloneable.class
                || expectClass == Closeable.class
                || expectClass == EventListener.class
                || expectClass == Iterable.class
                || expectClass == Collection.class
                ) {
            expectClassFlag = false;
        } else {
            expectClassFlag = true;
        }
    }
```

判断期望类，此时期望类为false。往下走的代码中，autoCloseable 满足不在白名单内,不在黑名单内，autoTypeSupport没有开启，expectClassFlag为null

其中：

A.计算哈希值进行内部白名单匹配

B.计算哈希值进行内部黑名单匹配

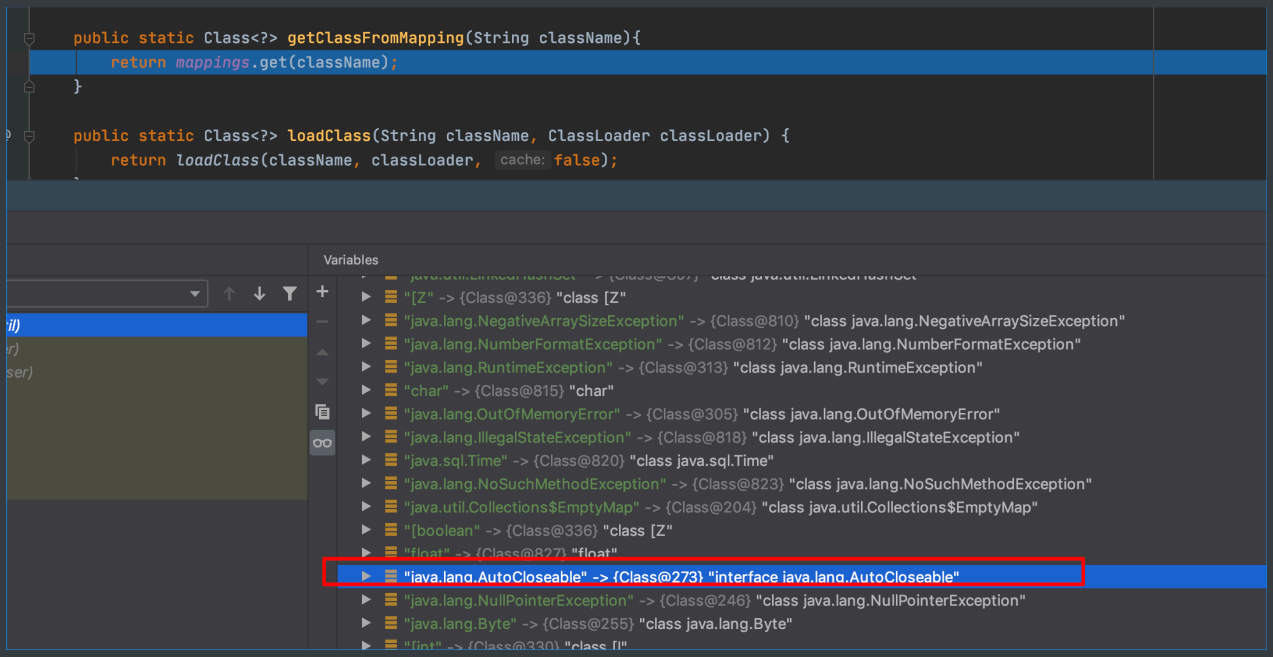
C.非内部白名单且开启autoTypeSupport或者是期望类的，进行hash校验白名单acceptHashCodes、黑名单denyHashCodes。如果在acceptHashCodes内则进行加载(defaultClassLoader),在黑名单内则抛出 autoType is not support。

```
    clazz = TypeUtils.getClassFromMapping(typeName);
```

满足条件C后来到clazz的赋值，解析来的代码中对clazz进行了各种判断

```
    clazz = TypeUtils.getClassFromMapping(typeName);
```

从明文缓存中取出autoCloseable赋值给 clazz




```
clazz = TypeUtils.getClassFromMapping(typeName);

if (clazz == null) {
    clazz = deserializers.findClass(typeName);
}

if (clazz == null) {
    clazz = typeMapping.get(typeName);
}

if (internalWhite) {
    clazz = TypeUtils.loadClass(typeName, defaultClassLoader, true);
}

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

    return clazz;
```

当clazz不为空时，expectClassFlag为空不满足条件，返回clazz,至此，第一次的checkAutoType检查完毕。

```
ObjectDeserializer deserializer = config.getDeserializer(clazz);
Class deserClass = deserializer.getClass();
if (JavaBeanDeserializer.class.isAssignableFrom(deserClass)
        && deserClass != JavaBeanDeserializer.class
        && deserClass != ThrowableDeserializer.class) {
    this.setResolveStatus(NONE);
} else if (deserializer instanceof MapDeserializer) {
    this.setResolveStatus(NONE);
}
Object obj = deserializer.deserialze(this, clazz, fieldName);
return obj;
```

将检查完毕的autoCloseable进行反序列化，该类使用的是JavaBeanDeserializer反序列化器，从MapDeserializer中继承

```
public <T> T deserialze(DefaultJSONParser parser, Type type, Object
fieldName) {
    return deserialze(parser, type, fieldName, 0);
}

public <T> T deserialze(DefaultJSONParser parser, Type type, Object
fieldName, int features) {
    return deserialze(parser, type, fieldName, null, features, null);
}//进入后代码如下
```

```
if ((typeKey != null && typeKey.equals(key))
        || JSON.DEFAULT_TYPE_KEY == key) {
    lexer.nextTokenWithColon(JSONToken.LITERAL_STRING);
    if (lexer.token() == JSONToken.LITERAL_STRING) {
        String typeName = lexer.stringVal();
        lexer.nextToken(JSONToken.COMMA);

        if (typeName.equals(beanInfo.typeName)||
parser.isEnabled(Feature.IgnoreAutoType)) {
        // beanInfo.typeName是autoCloseable ，但IgnoreAutoType没有开启
          if (lexer.token() == JSONToken.RBRACE) {
                lexer.nextToken();
                break;
            }
            continue;
        }//不满足条件所以这块代码被跳过了
```

JSON.DEFAULT_TYPE_KEY 为@type ,并给它赋值传入的key @type ,将第二个类也就是这次 的gadget传入

```
if (deserializer == null) {
    Class<?> expectClass = TypeUtils.getClass(type);
    userType = config.checkAutoType(typeName, expectClass,
lexer.getFeatures());
    deserializer = parser.getConfig().getDeserializer(userType);
}
```

期望类在这里发生了变化，expectClass的值变为java.lang.AutoCloseable，typeName为gadget，

```
boolean jsonType = false;
        InputStream is = null;
        try {
            String resource = typeName.replace('.', '/') + ".class";
            if (defaultClassLoader != null) {
                is = defaultClassLoader.getResourceAsStream(resource);
            } else {
                is =
ParserConfig.class.getClassLoader().getResourceAsStream(resource);
              //开了一个class文件的输入流
            }
            if (is != null) {
                ClassReader classReader = new ClassReader(is, true);//new
reader工具
                TypeCollector visitor = new TypeCollector("<clinit>", new
Class[0]);
                classReader.accept(visitor);
                jsonType = visitor.hasJsonType();
            }
        } catch (Exception e) {
            // skip
        } finally {
            IOUtils.close(is);//关闭流 JarURLConnection$JarURLInputStream
        }
```

来到JSONType注解，取typename gadget转换变为路径，resource通过将 "." 替换为"/"得到路径 。其实已经开始读取gadget了，它本意应该是加载AutoCloseable。

```
public ClassReader(InputStream is, boolean readAnnotations) throws
IOException {
```

```
        this.readAnnotations = readAnnotations;

        {
            ByteArrayOutputStream out = new ByteArrayOutputStream();
            byte[] buf = new byte[1024];
            for (; ; ) {
                int len = is.read(buf);
                if (len == -1) {
                    break;
                }

                if (len > 0) {
                    out.write(buf, 0, len);
                }
            }
            is.close();
            this.b = out.toByteArray();
        }
```

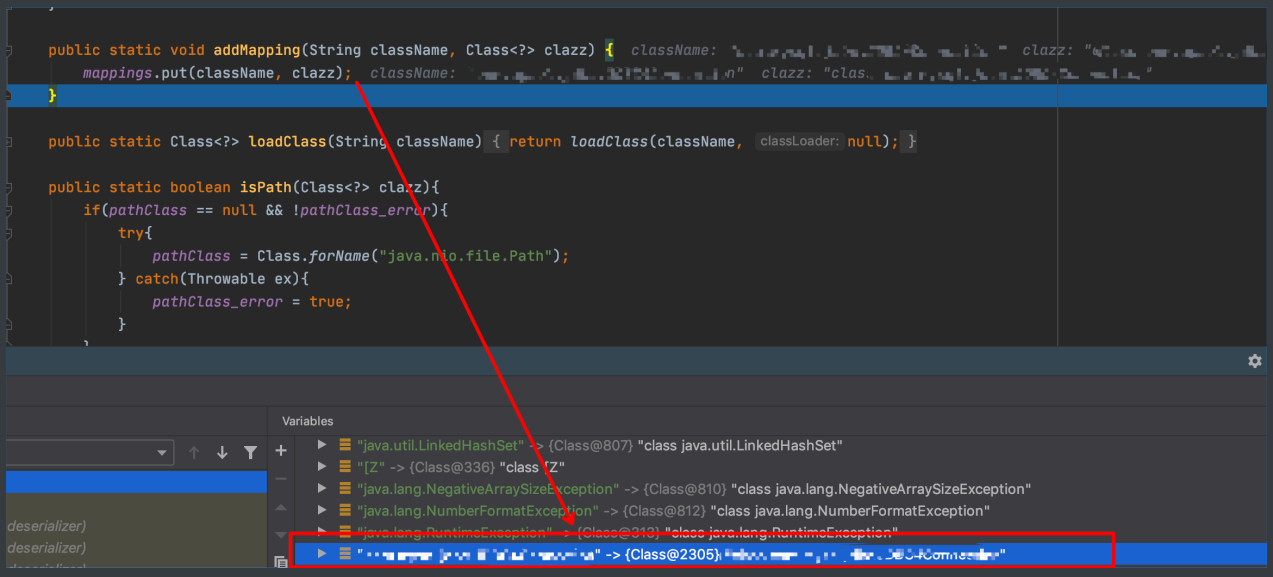
可以看到这里有读取文件的功能。所以之前网传的POC可能是利用这里这个特性(?)留意一下以后研究...

```
    if (autoTypeSupport || jsonType || expectClassFlag) {
        boolean cacheClass = autoTypeSupport || jsonType;
        clazz = TypeUtils.loadClass(typeName, defaultClassLoader, cacheClass);
      //开始加载gadget
    }
```

```
    if (expectClass != null) {
        if (expectClass.isAssignableFrom(clazz)) {//判断里面的类是否为继承类
            TypeUtils.addMapping(typeName, clazz);
            return clazz;
        } else {
            throw new JSONException("type not match. " + typeName + " -> " +
expectClass.getName());
        }
    }
```

isAssignableFrom()这个方法用于判断里面的类是否为继承类，当利用了java.lang.AutoCloseable这个方法去攻击fastjson，那么后续反序列化的链路一定是要继承于该类的子类。(daybr4ak大佬的博客写的)

TypeUtils.addMapping(typeName, clazz)这一步成功把gadget加入缓存中并返回被赋值gadget的clazz.



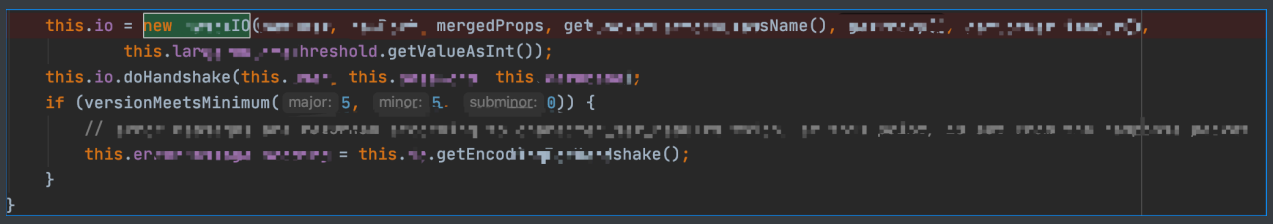
checkAutoType正式检查完毕，此时用deserializer = parser.getConfig().getDeserializer(userType);
userType既gadget进行反序列化。

```
    private void xxTryOnly(boolean isXXXXeconnect, Properties mergedProps)
    throws
      XXXException {
        Exception connectionNotEstablishedBecause = null;

        try {

            coreConnect(mergedProps);
            this.connectionId = this.io.getThreadId();
            this.isClosed = false;
```

进入coreConnect()




```
try {
    this.connection = this.socketFactory.connect(this.host, this.port, props);

    if (socketTimeout != 0) {
        try {
            this.mysqlConnection.setSoTimeout(socketTimeout);
        } catch (Exception ex) {
            /* Ignore if the platform does not support it */
        }
    }
```

在这里进行连接。至此漏洞利用完结。

三、参考考链接

https://b1ue.cn/archives/348.html

https://y4er.com/post/fastjson-bypass-autotype-1268/