Apache Tika 任意代码执行详细分析Poc—【CVE-2016-6809】

orich1 / 2017-12-28 09:48:00 / 浏览数 3202

---

seclists.org 在 2016.11.10 发布了一条消息，通告了一个存在于 Apache Tika 组件中的任意代码执行漏洞，编号为 cve-2016-6809 ，影响的范围是1.6-1.13版本
http://seclists.org/bugtraq/2016/Nov/40
tika 也是可以和 solr 整合到一起工作的
solr 影响的版本是：
Solr 5.0.0 to 5.5.4
Solr 6.0.0 to 6.6.1
Solr 7.0.0 to 7.0.1

本来是准备找好poc开始分析的，结果网上搜了一圈，只有如下信息

```
Description: Apache Tika wraps the jmatio parser (https://github.com/gradusnikov/jmatio) to handle MATLAB files.  The parser u
```

大意是 tika 包装了 jmatio 这个解析器去处理 matlab 文件，但是呢这个解析器使用了 java 本身自带的反序列化工具去处理镶嵌在 matlab 文件中的 java objects，我们可以向文件中注入恶意代码，所以在tika解析matlab文件时造成了任意代码执行

尴尬的是，并没有poc更别说exp了
只能自己动手

简单看了下，发现有三个地方可以让程序跑起来：
tika-app 里的 TikaCLI 和 TikaGUI

tika-server 里的 TikaServerCli

其中 TikaCLI 和 TikaGUI 都是本地开启的图形化窗口程序，一个是客户端，一个是服务端
TikaServerCli 是开启的类似于web网站api，绑定在9998端口（当然可以自行修改），如下

为了方便分析，我们直接选取本地的图形化窗口就行，我这里选的是 TikaGUI 作为调试程序

看看它的按钮对应的功能

跟进 addMenuItem

如上图，openfile已经被设置为 ActionCommand 了，并且增添了对应的监听器

那么我们去看看 action 是怎么执行的，查看 actionPerformed

如果我们点击的是 File 下的 Open... 的话，那么就会相应的调用 openFile 去加载我们选择的文件，继续跟进

这个 metadata 贯穿全局，一直都会用到，这里是将文件内容读取后交给了 handleStream 处理

将 input 处理后，交给了 parse 函数
这里的 parser 是在TikaGUI的main函数中已经确定

也就是说这里调用的 parse 函数是 DigestingParser 中的 parse
跟进去看看

这里又调用了父类的 parse 函数，它是继承于 ParserDecorator 的，跟进去

调用了 parser.parse ，看下 parser 是怎么初始化的

这其实在 DigestingParser 中调用过

parser 恰好是 main 中的 AutoDectectParser ，那么这儿就可以确定调用的是 AutoDectectParser 中的 parse 函数，跟进去

tis 就是之前选择的文件内容，metadata 新加入了文件的 MIME 信息
继续跟进父类 CompositeParser 的 parse 函数

如上图，他首先会用 metadata 去实例化一个 parser，然后判断 parser 的类型，在 metadata中添加一个 X-Parsed-By hash，最后会调用 parser 的 parse 函数

但是第一次调用 getParser 的时候，默认会实例化一个名叫 DefaultParser 的类，然后调用它的 parse
函数，这个过程就不跟进了，处理结果是，又会调用CompositeParser 的parse函数，并且带入了 MIME 信息，所以第二次 getParser 的时候，会以 MIME 信息来实例化对应的 parser，由漏洞通报所述，对matlab文件的处理，就是 MatParser 了

此时 metadata 里的 X-Parsed-By：

跟进 MatParser 的 parse 函数

如上图中，先将 Content-Type 添加进了 metdata 中，tis 是提取的由源文件生成的 tmp 文件，然后将其信息带入 MatFileReader 的构造函数中，跟进去看看（这里的 MatFileReader 其实就是 jmatio 解析器的文件读取类）

emm...其中有很多步直接略过了，直接来到MatFileReader 类中 read 函数中（MatFileReader-> MatFileReader->this.read)

在 read 函数中发现现了类似读取的操作：

这里的 buf 是 matlab 文件流
继续跟进 readData 函数

```
private void readData(ByteBuffer buf) throws IOException {
   MatFileReader.ISMatTag tag = new MatFileReader.ISMatTag(buf);
   switch(tag.type) {
   case 14:
       int pos = buf.position();
       MLArray element = this.readMatrix(buf, true);
       int red;
       int toread;
       if (element != null && !this.data.containsKey(element.getName())) {
           this.data.put(element.getName(), element);
       } else {
           red = buf.position() - pos;
           toread = tag.size - red;
           buf.position(buf.position() + toread);
       }

       red = buf.position() - pos;
       toread = tag.size - red;
       if (toread != 0) {
           throw new MatlabIOException("Matrix was not red fully! " + toread + " remaining in the buffer.");
       }
       break;
   case 15:
       int numOfBytes = tag.size;
       if (buf.remaining() < numOfBytes) {
           throw new MatlabIOException("Compressed buffer length miscalculated!");
       }

       InflaterInputStream iis = new InflaterInputStream(new ByteBufferInputStream(buf, numOfBytes));
       byte[] result = new byte[1024];
       HeapBufferDataOutputStream dos = new HeapBufferDataOutputStream();

       try {
           int i;
           try {
               do {
                   i = iis.read(result, 0, result.length);
                   int len = Math.max(0, i);
                   dos.write(result, 0, len);
               } while(i > 0);
           } catch (IOException var23) {
               throw new MatlabIOException("Could not decompress data: " + var23);
           }
       } finally {
           iis.close();
           dos.flush();
       }

       ByteBuffer out = dos.getByteBuffer();
       out.order(this.byteOrder);

       try {
           this.readData(out);
           break;
       } catch (IOException var21) {
           throw var21;
       } finally {
           dos.close();
       }
   default:
```

```
        throw new MatlabIOException("Incorrect data tag: " + tag);
    }

}
```

ISMatTag 这个静态内部类很重要，因为 java 和 matlab 数据格式本身就不同，所以在java里需要读取 matlab 的数据的话，需要利用到Tag（标识），每用一次 ISMatTag ，就会读取文件中剩余部分的 Tag ，然后依次来判别更多的标识、数据格式等等

如上代码中，一进 readData 函数中，就获取了一次 Tag，然后根据其 type ，来进行不同的处理，其中 type 为 14
表示未压缩的数据，15表示已压缩的数据，如果是15，那么经过处理后，type会变成14，代码：

```
case 14:
        int pos = buf.position();
        MLArray element = this.readMatrix(buf, true);
```

这里将 buf 带入了 readMatrix 函数里，跟进去

```
private MLArray readMatrix(ByteBuffer buf, boolean isRoot) throws IOException {
    int[] flags = this.readFlags(buf);
    int attributes = flags.length != 0 ? flags[0] : 0;
    int nzmax = flags.length != 0 ? flags[1] : 0;
    int type = attributes & 255;
    int[] dims = this.readDimension(buf);
    String name = this.readName(buf);
    if (isRoot && !this.filter.matches(name)) {
        return null;
    } else {
        Object mlArray;
        MatFileReader.ISMatTag tag;
        MLStructure struct;
        int maxlen;
        int numOfFields;
        String[] fieldNames;
        String className;
        int i;
        MLArray fieldValue;
        switch(type) {
```

这里可以确定的是 isRoot 为 true，所以第一个 if 判断，我们进入了 else 中
不过 readFlags、readDimension、 readName 都是利用 ISMatTag 读取出来的
type 也是由 ISMatTag 的信息计算得出

然后就进入了 switch 中，这里就真正开始解析一个 Tag 后的数据流了

switch 中用到的 数值都在 MLArray 中

如上图，这都是 matlab 中的数据格式
我们注意一下 mxOPAQUE_CLASS 格式的，那么这个到底对应的是啥呢
这里我们思考一下，这里整个都是一个读取文件的操作，那么对应的，在 jmatio中，肯定也有写入文件的操作
那么我们去网上找找这个 jmatio 的用法
写matlab文件的操作：

这里就很明显了，double的二维数组，可以对应 MLDouble ,我们查看一下这个类

无论是调用的哪一个构造函数，都会最终调用父类 MLNumbericArray 的构造函数，并且type都是6，那么我们跟进去看看，这个 type 究竟是什么

又调用了父类 MLArray 中的构造函数，并且传递了 type，继续跟

存在了type变量中
回想一下之前所说，switch用到的 type 是和 MLArray 中的那一串常量有关系的
并且进入switch的type是由Tag计算得出，写入的时候得先指定 MLDboule 类型，读取的时候也应该先获取它的类型，然后再根据其类型进行具体的操作。
那么这里猜测 switch的type 就是指的 MLDouble 构造函数里的type，那么我们去查看一下 MLArray 有哪些继承类

有一个显眼的 MLJavaObject 类，看看

可以看见它的构造函数第一句就调用了父类的构造函数，并且将type设置为了 17
这一看就是将 java object 写入matlab文件的操作
那么这就和之前的对应起来了，mxOPAQUE_CLSS 对应的就是 17
现在继续跟进 readMatrix 函数，我们需要反序列化，肯定type是17，那么就去看看 case 17

className 从对应的 Tag 信息里获取，也是对应刚刚看到的MLJavaObject里的 className，arrName 对应 MLJavaObject 里的 name，用于方便 matlab 里调用（可以认为matlab里的变量名就是 arrName ）

然后又将 buf 传入了 readMatrix 函数里，不过这次将 isRoot 设置为false了，由这里可以得知，真正的序列化后的数据流是以 UInt8 的格式存储的，这里就不跟入了，原理和跟入 case 17 是一样的
得到content后，将其处理为 ObjectInputStream 类型，紧接着就进行反序列化操作了
如果 matlab文件的内容是我们精心构造的，就可以触发代码执行

因为有对应的写入matlab文件操作，所以想着是利用 jmatio 自带的写入操作去生成含有payload的文件的，结果仔细一看，他并没有实现 java object 类型写入，还有一些类型的写入同样没有实现，然后晃了一眼 jmatio，是 1.0 的，想去找找最新版本，看看有没有实现来着，结果同样的没有实现
那么，这个时候有两种办法：

1. 自己去实现它的 java object 类型写入
2. 用matlab直接生成一个 java object 然后存进一个mat文件里，但是这个洞的触发是因为是用的 jmatio 去解析 mat文件，所以不知道 matlab 生成的 mat 文件，他是否能准确识别

emmm，说来比较尴尬，因为比较懒所以不想搭matlab环境，然后尝试着自己去实现这个 java object的写入操作.....但是由于水平有限，只是分析出了该怎么写，然鹅写不出来.....
所以和小伙伴@0v0商量着用 matlab 来搞payload
经过简单测试，matlab里的int数组类型存入 .mat 文件后，在经过 jmatio 解析文件是ok的
所以着手payload的构造
所需工具：
matlab 2006b 以上版本，具体不知道，测试的时候用的是 2016

为了方便测试，我们就选择 commons-collections 爆的代码执行了，选择的是 3.2 版本

编写 payload：

instance 成员变量是为了 matlab 里方便进行存入 .mat 文件的操作，其他地方都一样
调用 seria 的时候，对象的 instance 变量就已经赋值成了 AnnotationInvocationHandler 了，那么在 matlab 里，存入这个 instance 的话， .mat 文件内容就是 instance 的序列化
（ matlab 生成 .mat 文件时，会自动将变量进行序列化操作 ）
写好 payload 后，将其打包成 jar 包，然后在 matlab 里设置一下，让其可以引用这个 jar 包

打开 matlab

此时已经在桌面上生成了 test.mat 文件，内容就是 h.instance
为了初步验证 test.mat 含有我们的payload，我们在 matlab 里进行load一次，去加载这个文件

成功弹了，说明payload确实存在的

现在我们去 tika 里试试，直接用本地gui吧
（ 注：需要自行添加 commons-collections ，tika 是没有使用这个的 ）

(附上 test.mat)

使用jmatio读写matlab数据文件：https://www.cnblogs.com/lz3018/p/5210443.html

test.rar (0.001 MB) 下载附件
点击收藏 | 0 关注 | 0
上一篇：PHPcms9.6.3存储型XSS 下一篇：Web狗要懂的内网端口转发
1. 4 条回复


alex 2017-11-28 20:54:22

就喜欢看技术的分析文章~

0 回复Ta


b5mali4 2017-12-04 11:06:37

牛逼啊，大师傅

0 回复Ta


来自外星人 2017-12-15 14:36:55

为了看附件给师傅写回复

0 回复Ta


小马安全 2017-12-29 08:20:23

学习学习，谢谢分享

0 回复Ta

登录 后跟帖

先知社区

现在登录

热门节点

技术文章

社区小黑板

目录

RSS 关于社区 友情链接 社区小黑板