目录

[1. 关于RMI 1](#_Toc35451488)

[2. JNDI注入和JdbcRowSetImpl利用链 4](#_Toc35451489)

[3. 远程利用FAQ 8](#_Toc35451490)

## 关于RMI

这一节主要介绍一下RMI的调用流程、RMI注册表以及动态加载类的概念。

#### 远程方法调用

远程方法调用是分布式编程中的一个基本思想。实现远程方法调用的技术有很多，比如：CORBA、WebService，这两种都是独立于编程语言的。而RMI（Remote Method Invocation）是专为Java环境设计的远程方法调用机制，远程服务器实现具体的Java方法并提供接口，客户端本地仅需根据接口类的定义，提供相应的参数即可调用远程方法。RMI依赖的通信协议为JRMP(Java Remote Message Protocol ，Java 远程消息交换协议)，该协议为Java定制，要求服务端与客户端都为Java编写。这个协议就像HTTP协议一样，规定了客户端和服务端通信要满足的规范。在RMI中对象是通过序列化方式进行编码传输的。

#### 远程对象

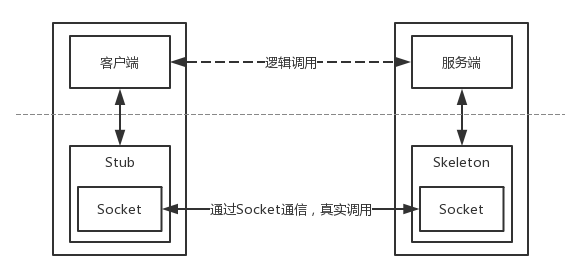
使用远程方法调用，必然会涉及参数的传递和执行结果的返回。参数或者返回值可以是基本数据类型，当然也有可能是对象的引用。所以这些需要被传输的对象必须可以被序列化，这要求相应的类必须实现 java.io.Serializable 接口，并且客户端的serialVersionUID字段要与服务器端保持一致。

任何可以被远程调用方法的对象必须实现 java.rmi.Remote 接口，远程对象的实现类必须继承UnicastRemoteObject类。如果不继承UnicastRemoteObject类，则需要手工初始化远程对象，在远程对象的构造方法的调用UnicastRemoteObject.exportObject()静态方法。如下：



注: IHello是客户端和服务端共用的接口（客户端本地必须有远程对象的接口，不然无法指定要调用的方法，而且其全限定名必须与服务器上的对象完全相同），HelloImpl是一个服务端远程对象，提供了一个sayHello方法供远程调用。它没有继承UnicastRemoteObject类或者实现java.rmi.Remote接口，而是在构造方法中调用了UnicastRemoteObject.exportObject()。

在JVM之间通信时，RMI对远程对象和非远程对象的处理方式是不一样的，它并没有直接把远程对象复制一份传递给客户端，而是传递了一个远程对象的Stub，Stub基本上相当于是远程对象的引用或者代理。Stub对开发者是透明的，客户端可以像调用本地方法一样直接通过它来调用远程方法。Stub中包含了远程对象的定位信息，如Socket端口、服务端主机地址等等，并实现了远程调用过程中具体的底层网络通信细节，所以RMI远程调用逻辑是这样的：



从逻辑上来看，数据是在Client和Server之间横向流动的，但是实际上是从Client到Stub，然后从Skeleton到Server这样纵向流动的。

Server端监听一个端口，这个端口是JVM随机选择的；

Client端并不知道Server远程对象的通信地址和端口，但是Stub中包含了这些信息，并封装了底层网络操作；

Client端可以调用Stub上的方法；

Stub连接到Server端监听的通信端口并提交参数；

远程Server端上执行具体的方法，并返回结果给Stub；

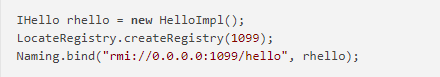
Stub返回执行结果给Client端，从Client看来就好像是Stub在本地执行了这个方法一样；

那怎么获取Stub呢？

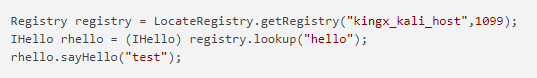
#### RMI注册表

Stub的获取方式有很多，常见的方法是调用某个远程服务上的方法，向远程服务获取存根。但是调用远程方法又必须先有远程对象的Stub，所以这里有个死循环问题。JDK提供了一个RMI注册表（RMIRegistry）来解决这个问题。RMIRegistry也是一个远程对象，默认监听在传说中的1099端口上，可以使用代码启动RMIRegistry，也可以使用rmiregistry命令。

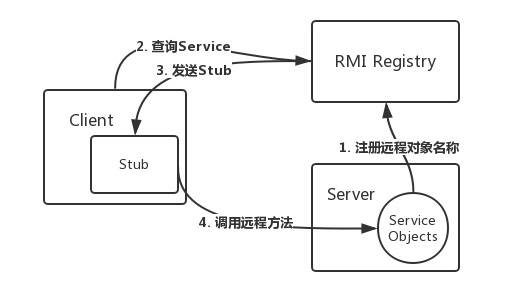
要注册远程对象，需要RMI URL和一个远程对象的引用：



LocateRegistry.getRegistry()会使用给定的主机和端口等信息本地创建一个Stub对象作为Registry远程对象的代理，从而启动整个远程调用逻辑。服务端应用程序可以向RMI注册表中注册远程对象，然后客户端向RMI注册表查询某个远程对象名称，来获取该远程对象的Stub。



使用RMI Registry之后，RMI的调用关系是这样的：



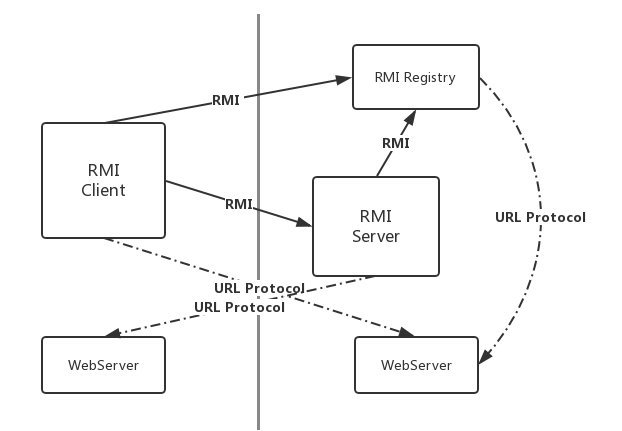
所以其实从客户端角度看，服务端应用是有两个端口的，一个是RMI Registry端口（默认为1099），另一个是远程对象的通信端口（随机分配的）。这个通信细节比较重要，真实利用过程中可能会在这里遇到一些坑。

#### 动态加载类

RMI核心特点之一就是动态类加载，如果当前JVM中没有某个类的定义，它可以从远程URL去下载这个类的class，动态加载的对象class文件可以使用Web服务的方式进行托管。这可以动态的扩展远程应用的功能，RMI注册表上可以动态的加载绑定多个RMI应用。对于客户端而言，服务端返回值也可能是一些子类的对象实例，而客户端并没有这些子类的class文件，如果需要客户端正确调用这些子类中被重写的方法，则同样需要有运行时动态加载额外类的能力。客户端使用了与RMI注册表相同的机制。RMI服务端将URL传递给客户端，客户端通过HTTP请求下载这些类。

这个概念比较重要，JNDI注入的利用方法中也借助了动态加载类的思路。

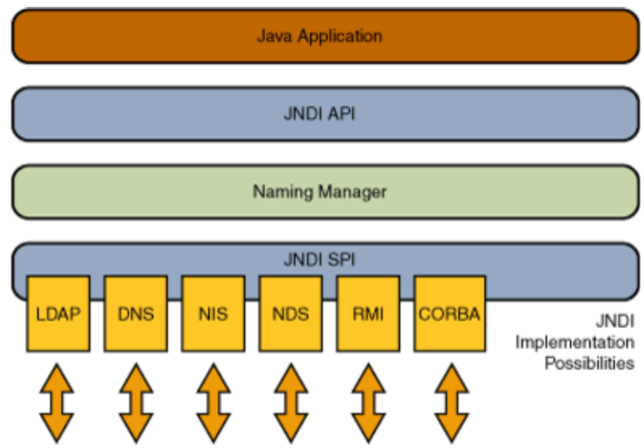
这里涉及到的角色：客户端、RMI注册表、远程对象服务器、托管class文件的Web服务器可以分别位于不同的主机上：



## JNDI注入和JdbcRowSetImpl利用链

#### 关于JNDI

简单来说，JNDI (Java Naming and Directory Interface) 是一组应用程序接口，它为开发人员查找和访问各种资源提供了统一的通用接口，可以用来定位用户、网络、机器、对象和服务等各种资源。比如可以利用JNDI在局域网上定位一台打印机，也可以用JNDI来定位数据库服务或一个远程Java对象。JNDI底层支持RMI远程对象，RMI注册的服务可以通过JNDI接口来访问和调用。



JNDI接口在初始化时，可以将RMI URL作为参数传入，而JNDI注入就出现在客户端的lookup()函数中，如果lookup()的参数可控就可能被攻击。



注：InitialContext 是一个实现了 Context接口的类。使用这个类作为JNDI命名服务的入口点。创建InitialContext 对象需要传入一组属性，参数类型为java.util.Hashtable或其子类之一。

#### 利用JNDI References进行注入

在JNDI服务中，RMI服务端除了直接绑定远程对象之外，还可以通过References类来绑定一个外部的远程对象（当前名称目录系统之外的对象）。绑定了Reference之后，服务端会先通过Referenceable.getReference()获取绑定对象的引用，并且在目录中保存。当客户端在lookup()查找这个远程对象时，客户端会获取相应的object factory，最终通过factory类将reference转换为具体的对象实例。

整个利用流程如下：

目标代码中调用了InitialContext.lookup(URI)，且URI为用户可控；

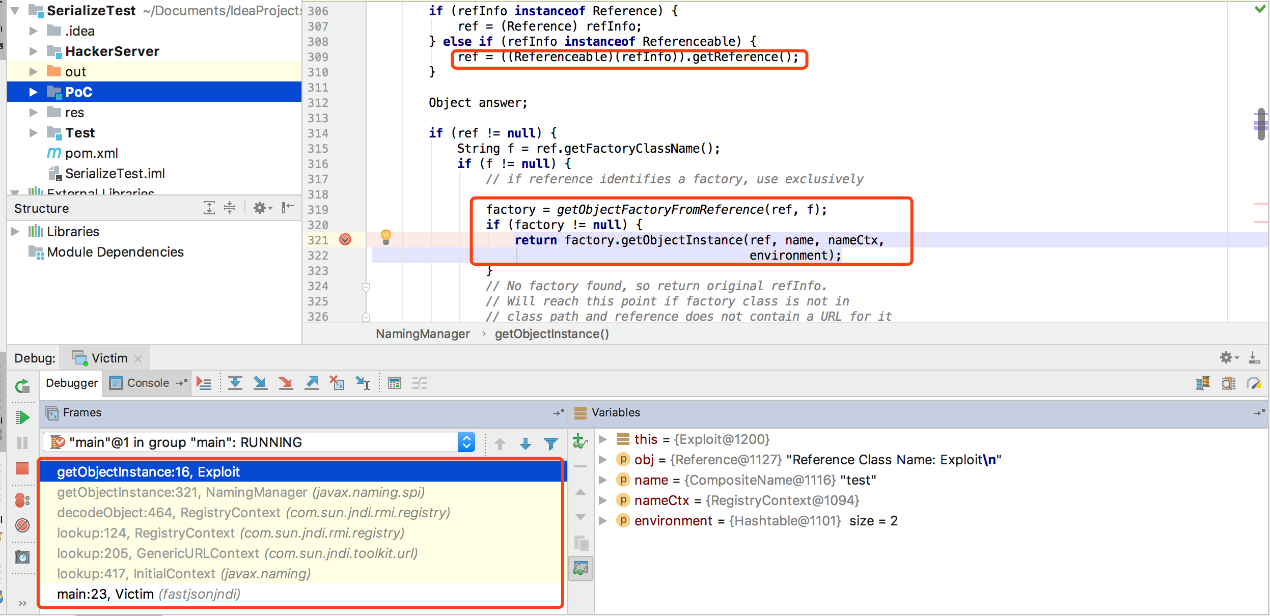
攻击者控制URI参数为恶意的RMI服务地址，如：rmi://hacker\_rmi\_server//name；

攻击者RMI服务器向目标返回一个Reference对象，Reference对象中指定某个精心构造的Factory类；

目标在进行lookup()操作时，会动态加载并实例化Factory类，接着调用factory.getObjectInstance()获取外部远程对象实例；

攻击者可以在Factory类文件的构造方法、静态代码块、getObjectInstance()方法等处写入恶意代码，达到RCE的效果；

在这里，攻击目标扮演的相当于是JNDI客户端的角色，攻击者通过搭建一个恶意的RMI服务端来实施攻击。我们跟入lookup()函数的代码中，可以看到JNDI中对Reference类的处理逻辑，最终会调用NamingManager.getObjectInstance()：



调用链：

-> RegistryContext.decodeObject()

-> NamingManager.getObjectInstance()

-> factory.getObjectInstance()

Tips：JNDI查找远程对象时InitialContext.lookup(URL)的参数URL可以覆盖一些上下文中的属性，比如：Context.PROVIDER\_URL。

Spring框架的spring-tx.jar中的JtaTransactionManager.readObject()中就存在这个问题，当进行对象反序列化的时候，会执行lookup()操作，可以进行JNDI注入。

Matthias Kaiser(matthias\_kaiser)发现com.sun.rowset.JdbcRowSetImpl类的execute()也可以触发JNDI注入利用，调用过程如下：

-> JdbcRowSetImpl.execute()

-> JdbcRowSetImpl.prepare()

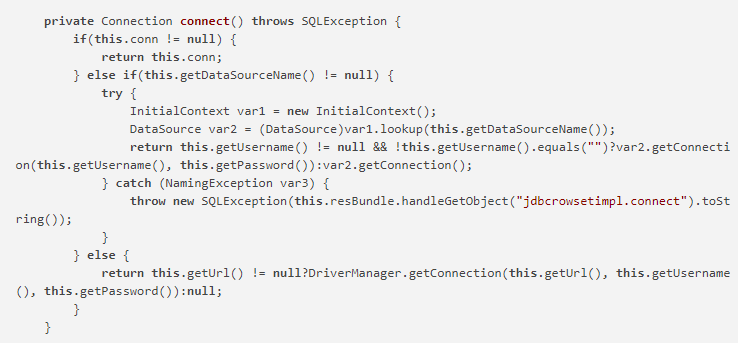
-> JdbcRowSetImpl.connect()

-> InitialContext.lookup(dataSource)

#### 2.3 FastJson反序列化利用

根据FastJson反序列化漏洞原理，FastJson将JSON字符串反序列化到指定的Java类时，会调用目标类的getter、setter等方法。

JdbcRowSetImpl类的setAutoCommit()会调用connect()函数，connect()函数如下：



connect()会调用InitialContext.lookup(dataSourceName)，这里的参数dataSourceName是在setter方法setDataSourceName(String name)中设置的。所以在FastJson反序列化漏洞过程中，我们可以控制dataSourceName的值，也就是说满足了JNDI注入利用的条件。利用Payload如下：



攻击者的服务端需要启动一个RMI Registry，并且绑定一个Reference远程对象，同时设置一个恶意的factory类。



同时启动一个WebServer提供Exploit.class下载。恶意代码可以放在构造方法中，也可以放在getObjectInstance(）方法中：

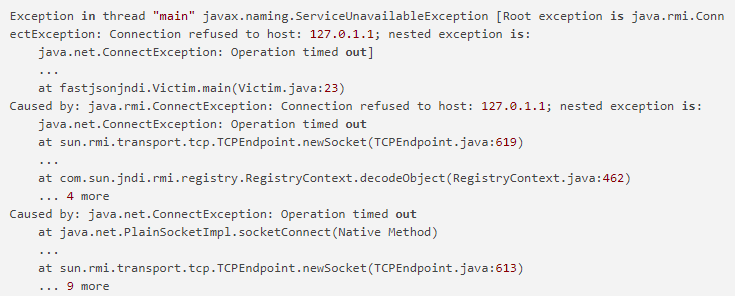


## 远程利用FAQ

网上很多PoC都是在本地测试的，然而在远程利用过程中可能会遇到一些坑，直接会导致利用失败，比如可能会遇到Timeout的错误。

#### 为什么远程利用会出现Timeout？

使用JNDI注入Payload进行利用时，有时候发现目标确实反连到我们的RMI服务器了，却没有去下载WebServer上的恶意class文件。我们在局域网内使用Kali作为攻击者RMI服务器，复现一下攻击过程，往往会看到类似这样的Timeout的错误提示：

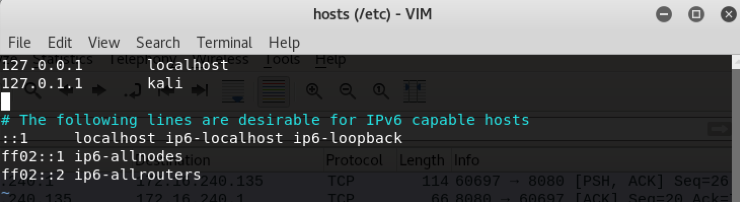


为什么会超时呢？

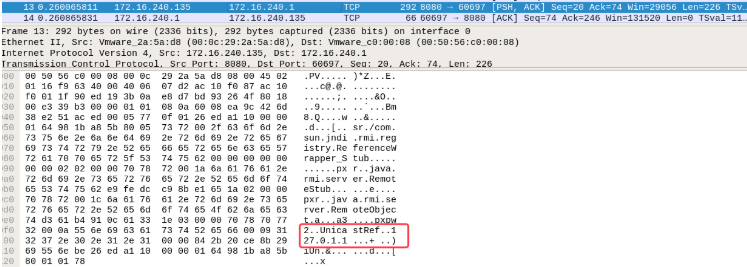
其实如我们在第一小节所说，启动了RMI Registry的服务端有两个端口，一个是RMI Registry监听端口，另一个是远程对象的通信端口。而远程对象通信端口是系统随机分配的，远程对象的通信Host、Port等信息由RMI Registry传递给客户端，通信Host的默认值是服务端本地主机名对应的IP地址。

参考链接：<https://docs.oracle.com/javase/7/docs/technotes/guides/rmi/javarmiproperties.html>

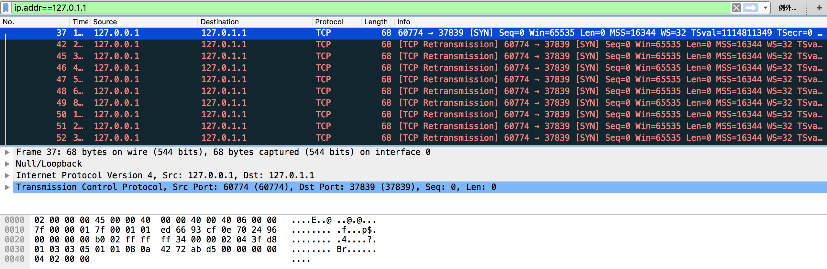
所以当服务器有多张网卡，或者/etc/hosts中将主机名指向某个内网IP的时候，RMI Registry默认传递给客户端的通信Host也就是这个内网IP地址，远程利用时自然无法建立通信。Kali默认的hosts文件如下：



可以看到，默认情况下kali主机名是解析到 127.0.1.1 了。我们通过抓包可以还原这个通信细节：



注：攻击者RMI服务端向目标发送远程对象的定位信息



注：目标向攻击者的远程对象发起请求

定位到问题解决起来就简单了。可以把/etc/hosts中指向内网IP的记录删除或者指向外网IP，也可以在攻击者的RMI服务端通过代码明确指定远程对象通信Host IP：



或者在启动RMI服务时，通过启动参数指定 java.rmi.server.hostname 属性：



1. References：

https://www.freebuf.com/column/189835.html