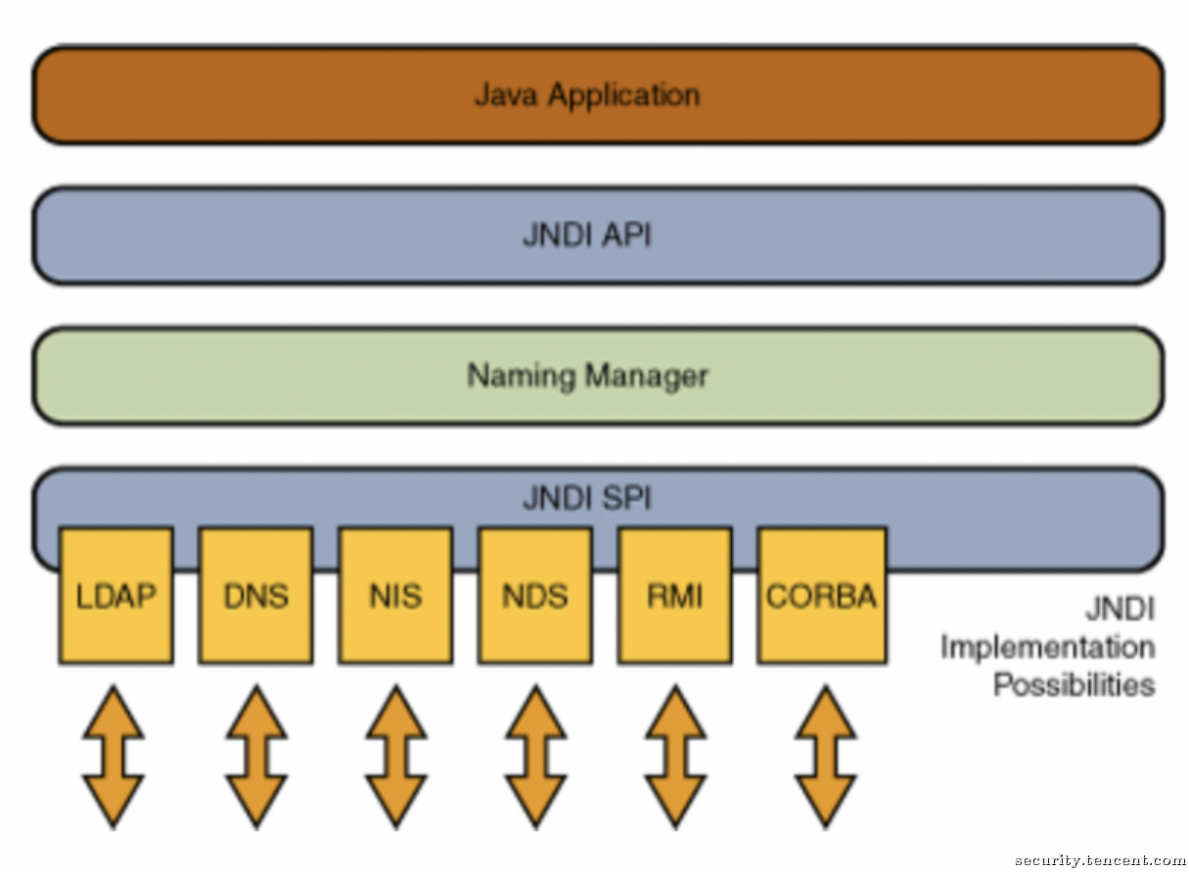
关于JNDI

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支持多种命名和目录提供程序（Naming and Directory Providers），RMI注册表服务提供程序（RMI Registry Service Provider）允许通过JNDI应用接口对RMI中注册的远程对象进行访问操作。将RMI服务绑定到JNDI的一个好处是更加透明、统一和松散耦合，RMI客户端直接通过URL来定位一个远程对象，而且该RMI服务可以和包含人员，组织和网络资源等信息的企业目录链接在一起。



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。

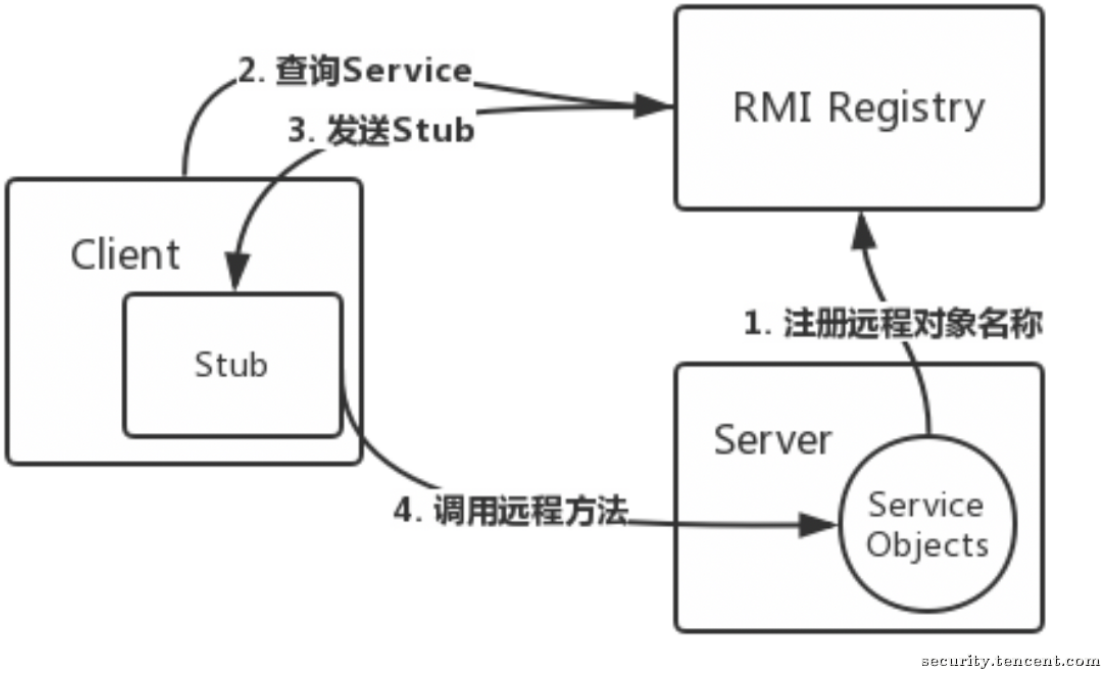
要注册远程对象，需要RMI URL和一个远程对象的引用。

```
IHello rhello = new HelloImpl();
LocateRegistry.createRegistry(1099);
Naming.bind("rmi://0.0.0.0:1099/hello", rhello);
```


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。

```
Registry registry = LocateRegistry.getRegistry("kingx_kali_host",1099);
IHello rhello = (IHello) registry.lookup("hello");
rhello.sayHello("test");
```


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。

利用JNDI References进行注入（CVE-2017-5645）

我们来到JNDI注入的核心部分，关于JNDI注入，@pwntester在BlackHat上的讲义中写的已经很详细。我们这里重点讲一下和RMI反序列化相关的部分。接触过JNDI注入的同学可能会疑问，不应该是RMI服务器最终执行远程方法吗，为什么目标服务器lookup()一个恶意的RMI服务地址，会被执行恶意代码呢？

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：

1. 目标代码中调用了InitialContext.lookup(URI)，且URI为用户可控；
2. 攻击者控制URI参数为恶意的RMI服务地址，如：rmi://hacker_rmi_server//name；
3. 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；
4. 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；
5. 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()：